Software-Qualitätssicherung durch Automatisierung

Ein modellbasierter Ansatz

vom Fachbereich Informatik der Technischen Universität Kaiserslautern
zur Verleihung des akademischen Grades
Doktor-Ingenieur (Dr.-Ing.)
genehmigte Dissertation

von

Dipl.-Inform. Andreas Metzger

Dekan des Fachbereichs: Professor Dr. Hans Hagen

Promotionskommission
Vorsitzender: Professor Dr. Klaus Madlener
Berichterstatter: Professor Dr. Gerhard Zimmermann
Juniorprofessor Dr. Andreas Rausch

Tag der wissenschaftlichen Aussprache: 17. Dezember 2004

Kaiserslautern, 2004
D 386

Danksagung

Die vorliegende Dissertation entstand während meiner Tätigkeit als wissenschaftlicher Mitarbeiter in der Arbeitsgruppe „VLSI Entwurf und Architektur“ am Fachbereich Informatik der Universität (jetzt TU) Kaiserslautern und im Teilprojekt D1 des Sonderforschungsbereichs 501 „Entwicklung großer Systeme mit generischen Methoden“.

Von ganzem Herzen danke ich Prof. Dr. Gerhard Zimmermann, der mir als Leiter der Arbeitsgruppe nicht nur die Möglichkeit zur Anfertigung meiner Dissertation gab, sondern mir auch als „Doktorvater“ stets ein kompetenter und kritischer Diskussionspartner war. Auch konnte ich von ihm sehr viel über wissenschaftliche Herangehensweisen (vor allem das Experimentieren) lernen, was mich auf meiner wissenschaftlichen Laufbahn stets begleiten wird.

Bedanken möchte ich mich bei den Mitgliedern der Promotionskommission: bei Professor Dr. Gerhard Zimmermann und Juniorprofessor Dr. Andreas Rausch für die Begutachtung der Arbeit und deren konstruktive Kritik und bei Professor Dr. Klaus Madlener für die freundliche Übernahme des Vorsitzes.

Ein weiterer Dank gilt den Kollegen der Arbeitsgruppe für die produktiven Diskussionen, speziell Jan Peter Riegel für seine von mir häufig in Anspruch genommenen Fähigkeiten als UNIX-Systemadministrator und insbesondere Stefan Queins, dessen Freundschaft und wissenschaftliche Kritik mich über Jahre in der Arbeitsgruppe begleitete. Ohne seine grundlegenden Arbeiten zur PROBAnD-Methode hätte ich viele der Ergebnisse nicht erzielen können.

Wesentlich zum Erfolg dieser Arbeit trugen auch die Studenten Wolfgang Hanke, Marc Krämer, Thomas Kuhn, Dominik Rehm, Christian Walke und Christian Webel bei, die als wissenschaftliche Hilfskräfte und Projekt- bzw. Diplomarbeiter viele der in dieser Arbeit verwendeten Werkzeuge implementierten bzw. an Fallstudien teilnahmen.

Besonders dankbar bin ich meiner Mutter, die auch in schweren Zeiten alles daran setzte, dass ich meine beruflichen und privaten Ziele erreichen konnte.

Meiner Lebensgefährtin Maria Reinhardt danke ich vor allem für Ihre Geduld, die ich so häufig strapazierte, wenn ich wieder einmal am Wochenende in meine Arbeit vertieft war.

Andreas Metzger
Kaiserslautern, im Dezember 2004

Kurzfassung

In dieser Arbeit wird gezeigt, wie durch eine *Automatisierung von Software-Entwicklungsaktivitäten* sowohl Effizienz- als auch Qualitätsgewinne erzielt und komplexe Aktivitäten beherrschbar gemacht werden können.

Dazu wird zunächst eine solide Basis für eine *modellbasierte Software-Entwicklung* geschaffen. Nach der Identifikation der Probleme der bisher üblicherweise eingesetzten *Metamodellierung* wird eine *verbesserte Multiebenenmodellierung* vorgeschlagen, welche die explizite Angabe der Instanziierbarkeit (Tiefe und Automatismus der Instanziierung) der Modellelemente erlaubt und damit eine deutliche Vereinfachung und bessere Verständlichkeit der Metamodelle ermöglicht.

Zur operationalen Beschreibung von *Modelltransformationen* im Kontext dieser Multiebenenmodellierung wird sodann die *Aktionssprache AL++* konzipiert. Insbesondere durch die Einführung von Syntaxelementen für die Handhabung von Relationen und Attributen und die Aufnahme von Reflexionskonstrukten in die Sprache AL++ werden Transformationen kompakt und generisch beschreibbar.

Anwendung finden diese Ansätze in der modellbasierten Entwicklung *reaktiver Systeme*. Dazu wird eine existierende Entwicklungsmethode erweitert, um eine durchgängige Automatisierung realisieren zu können. Die wichtigste Erweiterung ist dabei die *modifizierte Automatenmodellierung*, bei welcher erweiterte Endliche Automaten durch die Komposition getrennt modellierter Zustandsübergänge spezifiziert werden, was eine eindeutige Verfolgbarkeit zu den Anforderungen erlaubt.

Eingesetzt werden obige Techniken für die *statische Analyse* von Spezifikationen, wobei insbesondere die automatische *Detektion von Feature-Interaktionen* (also die Feststellung kritischer Wechselwirkungen zwischen Produktmerkmalen) in dieser Form erstmalig für den Bereich der reaktiven Systeme durchgeführt wird.

Daneben werden automatisierte *dynamische Analysen* auf der Basis generierter *Prototypen* betrachtet. Die Analyseergebnisse können automatisiert für die Modifikation und Neukonstruktion der Prototypen genutzt werden, womit Software-Entwicklungsexperimente vollständig in einem „*virtuellen Labor*“ durchgeführt werden können. Wichtigstes experimentelles Ergebnis ist, dass eine statische Parametrisierung einer „intelligenten“ Temperaturregelung möglich ist und daher eine Reduktion der notwendigen Produktmerkmale (und damit der Komplexität) erreicht werden kann.

In Fallstudien wird am Ende der Arbeit nachgewiesen, dass alleine durch die automatische Erzeugung von Entwicklungsdokumenten und die konsistente Änderung vorhergehender Dokumente durch die in dieser Arbeit implementierten Werkzeuge ein *Effizienzgewinn* von 54 % erreicht werden kann. Die Erstellung der eingesetzten Werkzeuge hätte sich dabei bereits nach zwei ähnlichen Projekten bezahlt gemacht.

Abstract

This thesis provides insights in how efficiency as well as quality improvements can be attained through the *automation of software development activities* and how such an automation helps in controlling the complexity of many of such activities.

To start with, a solid basis for *model-based software development* is provided. After problems of *metamodeling techniques* that are currently in use have been identified, an *improved multi-level modeling approach* is suggested, in which the instantiatability (depth and automatism of the instantiation) of each model element can be explicitly provided by the developer. This leads to a drastic reduction of a metamodel's complexity and thus supports an improved understandability.

To operationally specify *model transformations* in the context of this multi-level modeling approach, the *action language AL++* is designed. Especially through the introduction of syntactic elements for handling relations as well as attributes and the inclusion of reflection mechanisms into AL++, transformations can be specified in a compact and generic form.

These approaches are applied for the model-based development of *reactive systems*, for which an existing development method is extended, such that a thorough automation can be realized. The most important extension is the introduction of a modified technique for the modeling of extended finite state machines, which allows the specification of such automata by composing independently modeled state transitions, thus enabling the unambiguous traceability to requirements.

The above techniques are used for the *static analysis* of specifications, where especially the automatic *detection of feature interactions* (i.e., critical interrelationships between a system's features) is a novel contribution in the domain of reactive systems.

Besides that, automated *dynamic analyses* on the basis of generated *prototypes* are considered. The results of the analysis are used for automatically modifying and reconstructing the prototypes, thus allowing for the execution of software development experiments in a „*virtual laboratory*". The most important result is that a static parametrization of an "intelligent" temperature control system is feasible, thus reducing the number of features that have to be implemented, which in turn reduces the overall system complexity.

Finally, through the evaluation of several case studies it is shown that solely by automatically creating development documents and keeping those documents consistent with each other through tools, an *increase of efficiency* of 54% can be achieved. Also, it is estimated that the effort for creating the automation tools would already pay off after two consecutive projects of average size.

x

Inhaltsverzeichnis

1 Einleitung

A much more powerful technique for removal of complexity is to automate the handling of it. [...] *The history of software is replete with examples of powerful software tools that raised the overall level of development capability of people by allowing them to address a new set of problems.*

— Terry Bollinger, Philippe Gabrini, Louis Martin
[Guide to the Software Engineering Body of Knowledge „SWEBOK“, 2001]

Die Komplexität und die Größe moderner Software-Systeme steigen mit der zunehmenden Zahl an Funktionen, die von den Anwendern verlangt werden. Dieser Trend ist immer öfter auch in der Domäne der eingebetteten Mess-, Steuer- und Regelsysteme (oder der reaktiven Systeme) festzustellen. Diese Domäne reicht von Steuerungen für einfache Haushaltsgeräte über Mobiltelefone und moderne Kraftfahrzeugelektronik bis hin zu anspruchsvollen Prozess- und Gebäudeautomationssystemen. Wo im letzten Jahrzent im Automobilbereich noch Programmgrößen von wenigen hundert Kilobytes für die Sicherheitselektronik (wie z. B. ABS) und das „In-Car-Entertainment“ (Autoradio) üblich waren (siehe [Sch99a, S. 5]), erreichen diese Software-Systeme heute Code-Größen von über 60 Megabytes (siehe [Saa03]). Insbesondere moderne Gebäudeautomationssysteme (vgl. [Kra97]), welche eine Vielzahl physikalischer Effekte zur optimalen Steuerung und Energieeffizienz nutzen, weisen eine enorme Größe und Komplexität auf. So besteht das Automationssystem für das „Burj Al Arab“ in Dubai – eines der modernsten Hotels der Welt – aus 26 000 einzelnen Datenpunkten (Sensoren und Aktuatoren, siehe [BCG01]).

Diese Zunahme der Komplexität und Größe von Software-Artefakten stellt ein erhebliches Problem für die Entwicklung solcher Systeme dar, da die Entwicklungsmethoden und -techniken oft nicht skalieren. Dies bedeutet, dass wenn die Komple-

xität eine gewisse Schranke überschreitet, der Aufwand zur Erstellung des Software-Produkts in einem erheblichen (nichtlinearen) Maße ansteigt. Als Folgen dieses Effekts leidet die Qualität des fertigen Produkts – es bleibt nicht genügend Zeit für die Qualitätssicherung – oder wird ein Produkt gar nicht erst fertiggestellt.

Die Frage die sich also stellt ist:

„Wie erreicht man, dass die Software-Entwicklung trotz der Zunahme von Komplexität und Größe der zu erstellenden Artefakte skaliert?“

Die Antwort, welche die vorliegende Arbeit liefert, ist:

„Durch eine *Automatisierung* von Software-Entwicklungsaktivitäten erreicht man einen Effizienzgewinn und die Beherrschbarkeit komplexer Aktivitäten, was eine Skalierbarkeit der Software-Entwicklung ermöglicht.“

Diese Skalierbarkeit wird an folgenden Punkten, welche die Effekte einer Automatisierung aufzeigen, deutlich:

1. Viele für einen Menschen schwierige Aktivitäten können sehr einfach von einer „*Maschine*“ durchgeführt werden. Eines der augenscheinlichsten Beispiel ist die Verwendung eines (optimierenden) Hochsprachen-Compilers an Stelle der manuellen Übersetzung eines Programms einer höheren Programmiersprache in die Maschinensprache des Rechners. Neben solchen Übersetzungsaktivitäten können aber auch komplexere Analyseaufgaben sehr effizient von einem Algorithmus gelöst werden.
2. Durch den *Zeitgewinn*, den man durch die Automatisierung von Aktivitäten in der Software-Entwicklung erreicht, kann bei gleichem Budget mehr Zeit für Qualitätssicherungsmaßnahmen (z.B. Prototyping und Inspektion) aufgewendet werden, was sich in einer höheren Produktqualität äußert.
3. Durch eine Automatisierung „stupider“ bzw. anspruchsvoller Aktivitäten wird die Wahrscheinlichkeit von *Flüchtigkeitsfehlern* bzw. „echten“ Entwicklungsfehlern geringer und dadurch die Gesamtfehlerzahl im Produkt reduziert.
4. Durch die implizit in Automatisierungswerkzeugen vorhandene *Wiederverwendung* (z.B. durch die Erzeugung domänenspezifischer Artefakte) wird eine höhere Qualität erzielt, da wiederverwendete Artefakte (und Aktivitäten) i.d.R. gründlich verifiziert und validiert wurden.
5. Die Automatisierung von Messungen (z.B. die Berechnung von Produktmaßen) erlaubt die permanente (und kosteneffiziente) *Qualitätskontrolle* und Aufwandsabschätzung während der Projektdurchführung.
6. Werden Software-Artefakte (Modelle und Code) automatisch konsistent gehalten, erhöht sich die *Wartbarkeit* des Software-Produkts. Dies ist in unseren Augen eine ernstzunehmende Alternative zu den agilen Methoden, wo man auf viele Dokumente verzichtet, da eine manuelle Konsistenthaltung in der Praxis selten durchgeführt wird (siehe dazu auch [Pin02]).

Möglich wird eine Automatisierung von Entwicklungsaktivitäten durch heutzutage verfügbare mächtige Techniken und Werkzeuge, welche auf *Modellen* arbeiten. Damit wird die Spezifikation der Entwicklungsartefakte und die Erstellung der Automatisierungs-Tools in einer effizienten Weise ermöglicht. Ohne die kostengünstige Erstellung der Automatisierungswerkzeuge kann der oben angesprochene Effizienzgewinn nicht realisiert werden. Der in dieser Arbeit vorgestellte systematische Ansatz stellt eine mögliche effiziente Realisierung solcher Automatisierungswerkzeuge dar, was experimentell nachgewiesen und durch die Präsentation konkreter Automatisierungsalgorithmen unterstrichen wird.

Interessant ist, dass eine Automatisierung von Aktivitäten, die der Evolution von Artefakten dienen, nicht weit verbreitet ist. Unter Evolution verstehen wir ein Voranbringen der Produktfunktionalität und nicht der Implementierung dieser Funktionalität. Diese Tatsache liegt sicherlich darin begründet, dass Programmcode (auch der in einer höheren Programmiersprache) semantisch nicht reich genug für eine solche automatische Transformation ist. Erst mit der Einführung von Modellierungssprachen und von domänenspezifischen Modellen für einzelne Anwendungsbereiche oder -methoden kann solch ein Vorhaben gelingen. Ein wichtiger Beitrag unserer Arbeit ist, genau solche Transformationen (und auf dieser Basis auch höherwertige Analysen) zu ermöglichen und dies an Hand konkreter Aktivitäten und Fallstudien zu belegen.

Kurz zusammengefasst sind die wichtigsten Aspekte des in dieser Arbeit vorgestellten Ansatzes

1. der Fokus auf die automatische Modellevolution und -analyse und nicht auf die Implementierung von Modellen durch Abbildungen auf Code,
2. der durchgehende Einsatz einer objektorientierten Modellierung und einer Multiebenenbeschreibung (Meta-Modellierung) zur Realisierung mächtiger und generischer Aktivitäten,
3. der Einsatz von Modellen als primäre Artefakte, die während der Software-Entwicklung erzeugt, modifiziert und gewartet werden,
4. die Automatisierung von Aktivitäten durch eine operationale Sprache (ähnlich zu Java), was den Zugang zur Erstellung von Automatisierungswerkzeugen für Modellierer mit Programmierkenntnissen erleichtert, und
5. eine dokumentbasierte Repräsentation von Entwicklungsmodellen, welche die Parallelarbeit vieler Entwickler erlaubt.

Verwandte Arbeiten

Die vorliegende Arbeit gliedert sich in die Bereiche des automatisierten Software-Engineering und der modellbasierte Entwicklung von Software („*Model-Driven Development*“ oder *MDD*, siehe z. B. [Sel03]) ein. In diesen beiden Bereichen, die einen

großen Überscheidungsbereich aufweisen, wurden in den vergangenen Jahren viele interessante Forschungsergebnisse publiziert.

Um diese Ergebnisse den unseren differenziert gegenüberstellen zu können, werden diese in den jeweiligen Kapiteln diskutiert und erläutert, weshalb von einer globalen Vorstellung verwandter Arbeiten an dieser Stelle abgesehen wird.

Es sei angemerkt, dass die in dieser Arbeit vorgestellten Ergebnisse zum Teil bereits in [ZiM04], [Met04], [Met04a], [MeW03], [MeQ03], [Met03a], [MeQ02], [MeQ02a], [MMZ02], [MeQ01], [Met01], [Met99] und [Met98] veröffentlicht und im Rahmen von [Met03] und [QuM02] vorgetragen wurden.

Gliederung der Arbeit

Die vorliegende Arbeit besteht aus zwei Teilen. Im ersten Teil werden die Grundlagen für eine modellbasierte Automatisierung von Entwicklungsaktivitäten gelegt, die dann im zweiten Teil auf konkrete Aktivitäten zur Sicherung der Software-Qualität angewendet werden.

In **Kapitel 2** wird dazu zunächst der Modellbegriff geklärt und insbesondere auf Syntax und Semantik von Modellen eingegangen. Dazu gehört eine Diskussion, inwiefern der Formalisierungsgrad von Modellen zur Automatisierung beiträgt. Nach der Einführung von Grammatiken und Metamodellen zur Modellbeschreibung werden die verschiedenen Modelltypen einer Software-Entwicklungsmethode (Produktmodell, Prozessmodell, etc.) im Kontext der Automatisierung eingeführt. Dies beinhaltet auch eine Klassifikation verschiedener Typen von Modelltransformationen, welche feingranularer als andere in der Literatur vorgestellten Ansätze ist.

Aufbauend auf dem Modellbegriff von Kapitel 2 werden in **Kapitel 3** die Grundlagen der modellbasierten Automatisierung präsentiert. Dazu gehört zuerst eine Einführung der objektorientierten Modellierung (inklusive der strukturellen und verhaltensorientierten Konzepte). Dies wird dann durch Einführung weiterer Modellebenen zur Metamodellierung verallgemeinert, wobei die Probleme der vorgestellten „traditionellen“ Metamodellierung diskutiert werden.

In **Kapitel 4** wird daher eine verbesserte Multiebenenmodellierung vorgeschlagen, die viele der in Kapitel 3 vorgestellten Ansätze verallgemeinert und dadurch eine durchgängige Modellierung ohne die diskutierten Probleme ermöglicht. Im weiteren Verlauf dieses Kapitels wird auf Basis der verbesserten Metamodellierung dann gezeigt, wie man eine Modelltransformation, die letztlich eine Modellierungsaktivität realisiert, spezifizieren und dadurch automatisch durchführen kann. Ein wichtiger Beitrag ist hierbei der Vergleich zwischen deklarativen und operationalen Ansätzen zur Modelltransformation. Für letzteren Ansatz wird eine konkrete Technik, die Aktionssprache AL++, eingeführt. Dabei erfolgt eine Erweiterung ggü. bekannten Aktionssprachen im Hinblick auf die verbesserte Multiebenenmodellierung,

wodurch eine natürliche Realisierung von Reflexionsmechanismen und damit die kompakte Formulierung generischer Algorithmen möglich wird.

In **Kapitel 5** wird unser auf den eingeführten Konzepten (Multiebenenmodellierung und Aktionssprache) basierende systematische Ansatz zur Automatisierung von Entwicklungsaktivitäten erläutert. Dabei werden die Vorteile einer Arbeit auf Dokumenten ggü. der Verwendung eines zentralen Repositorys begründet und gezeigt, wie man systematisch mit solchen Modelldokumenten umgehen kann. Zu dieser Systematik gehört auch die weitere Klassifikation von Produktmodellen durch Einführung einer Meta-Ebene. Dadurch lassen sich grundlegende Konzepte auf dieser Meta-Ebene erläutern und dann für jede passende Entwicklungsmethode instanziieren. Zu den Konzepten gehört das Parsen und Unparsen von Dokumenten, wobei durch den Parse-Schritt die vereinfachte Beschreibung der Modellevolution und -analyse auf abstrakten Datenstrukturen möglich wird. Schließlich wird gezeigt, wie alle Konzepte technisch umgesetzt wurden, um zu lauffähigen Werkzeugen zu gelangen. Hierzu war eine Abbildung der Aktionssprache auf eine Programmiersprache notwendig aber auch eine strukturierte Zuteilung von Verantwortlichkeiten auf einzelne Werkzeuge oder deren Komponenten. Am Ende von Kapitel 5 wird diese Arbeit von verwandten Arbeiten abgegrenzt und ähnliche Bereiche kurz abgerissen.

Mit **Kapitel 6** erreichen wir das Ende des ersten Teils der Arbeit. Hier wird die Entwicklungsmethode PROBAnD zur Spezifikation reaktiver Systeme vorgestellt und gezeigt, wie Aktivitäten dieser Methode automatisiert werden können. Dazu wird der Begriff des „reaktiven Systems“ präzise definiert und das Konzept der Verhaltensbeschreibung mittels Endlicher Automaten vorgestellt. Für die weitere Anwendung in der Arbeit wird eine modifizierte Automatenmodellierung eingeführt, bei welcher Automaten durch die Komposition getrennter Zustandsübergänge modelliert werden können, wodurch die Verfolgbarkeit zu den Anforderungen (also die Nachvollziehbarkeit) gewährleistet bleibt.

Der zweite Teil der Arbeit widmet sich der Anwendung der Automatisierungstechnik bzw. deren Einsatz für die in Kapitel 6 vorgestellte Entwicklungsmethode. Dazu wird in **Kapitel 7** zunächst allgemein auf Maßnahmen zur Qualitätssicherung eingegangen und interne von externen Qualitätseigenschaften abgegrenzt. Desweiteren wird gezeigt, wie die Qualitätssicherung durch konstruktive Maßnahmen unterstützt werden kann.

In **Kapitel 8** widmen wir uns den internen Qualitätseigenschaften. Dazu wird zunächst eine automatisierte Realisierung der Konsistenzprüfung und -herstellung von Entwicklungsdokumenten vorgestellt. Daran anschließend wird eine sehr komplexe Form der Modellanalyse vorgestellt, die der Detektion von Feature-Interaktionen (erwünschten und unerwünschten Abhängigkeiten zwischen Produktmerkmalen) dient. Eine manuelle Durchführung einer solchen Analyse würde nicht skalieren. Es wird daher gezeigt, wie hier durch die Implementierung von AL++-

Algorithmen ein enormer Effizienzgewinn verzeichnet werden kann. Eine letzte Anwendung unserer Automatisierungstechnik, die in diesem Kapitel vorgestellt wird, ist die automatischen Berechnung von Maßen. Dazu werden einige ausgewählte Beispielalgorithmen vorgestellt und zum Abschluss eine Visualisierung von Modelleigenschaften präsentiert.

Für eine ergänzende Sicherstellung der Qualität wird in **Kapitel 9** das automatisierte Prototyping reaktiver Systeme eingeführt. Dabei zählt zur Automatisierung nicht nur die frühe Erzeugung von Prototypen aus abstrakten Modellen (das Prototyping im engeren Sinne) sondern auch die automatische Instrumentierung solcher Prototypen. Schließlich kann auch die Aktivität der Anwendung von Prototypen (das Prototyping im weiteren Sinne) automatisiert werden. Dazu stellen wir eine Anwendung für das automatische Testen vor, die schließlich in einer automatisierten Ausführungsumgebung für Software-Experimente („virtuelles Labor“) mündet.

Die Effizienz des Ansatzes wird in **Kapitel 10** nachgewiesen. Dazu werden die Ergebnisse mit früheren Fallstudien verglichen und der Aufwand der Werkzeugerstellung genauer untersucht. Um vernünftige Messungen durchführen zu können, werden geeignete Produkt- und Prozessmaße eingeführt und versucht, diese so weit es möglich ist zu validieren. Außer den Vorteilen, die eine Automatisierung mit sich bringt, wird in Kapitel 10 auch auf deren Grenzen und Risiken hingewiesen.

In **Anhang A** sind die Konstrukte unserer Aktionssprache aus Kapitel 3 nochmals zusammengefasst. In **Anhang B** findet sich eine Zusammenstellung der in dieser Arbeit verwendeten Produktmodelle. Dazu gehören neben dem in Kapitel 5 definiertem Produktmetamodell auch das Produktmodell der in Kapitel 6 vorgestellten PROBAnD-Methode.

Vorbemerkung zum Sprachgebrauch

Im Text erfolgt die Bezeichnung weiblicher und männlicher Personen aus Gründen der Lesbarkeit und Übersichtlichkeit jeweils in der maskulinen Form. Mit allen verwendeten Personenbezeichnungen sind stets beide Geschlechter gemeint.

Die Orthographie entspricht den seit 1. August 1998 geltenden neuen amtlichen Rechtschreibregeln.

— Teil I —

Modellbasierte Software-Entwicklung

2 Modelle in der Software-Entwicklung

Die Verständnisvielfalt des Begriffes „Modell“ ist in der Literatur sehr ausgeprägt. Ursprung ist der lateinische Begriff „modulus“, welcher im 11. bis 14. Jahrhundert soviel wie Muster, Vorlage und Gussform bedeutete.

— Jochen Scheeg, Axel Hochstein
[Doktorandenseminar „Forschungsmethodik“, 2003]

Zu Beginn dieser Arbeit sollen die theoretischen und technischen Voraussetzungen für eine Automatisierung von Software-Entwicklungsaktivitäten diskutiert und mögliche Alternativen gegenübergestellt werden.

Dazu muss zunächst der zentrale Begriff des „*Modells*“ geklärt und die verschiedenen Arten von Modellen, die zur Beschreibung von Software bzw. deren Entwicklung eingesetzt werden, eingeführt werden. Die zwei wichtigsten Arten solcher Modelle, die für eine Automatisierung notwendig sind, sind das *Produktmodell* und das *Prozessmodell*. Erfüllen diese Modelle bestimmte Voraussetzungen, dann lassen sie sich für eine Automatisierung der Entwicklungsaktivitäten einsetzen. Dabei ist der Formalisierungsgrad dieser Modelle ausschlaggebend für das Maß an Automatisierung, das sich realisieren lässt.

Beide Modellarten werden in den folgenden Abschnitten behandelt und detailliert auf mögliche Klassen von Modelltransformationen eingegangen.

2.1 Modelle und Formalismen

Bevor eine Automatisierung überhaupt möglich wird, muss ein klares Verständnis des Software-Entwicklungsprozesses vorhanden sein, d.h. das Wissen über die Software-Entwicklung muss explizit gemacht werden. Daher werden *explizite Modelle* von den während der Entwicklung entstehenden *Artefakten* und den die Artefakte

bearbeitenden oder erzeugenden *Entwicklungsaktivitäten* benötigt. Nur durch explizite Modelle werden Entwicklungsaktivitäten wiederhol- und verfolgbar und letztendlich auch automatisierbar (siehe dazu auch [SPH02]).

Ein *Modell* beschreibt dabei eine *Abstraktion* des modellierten *Systems*. Unter dem Vorgang der Abstraktion kann man allgemein „das Heraussondern des unter einem bestimmten Gesichtspunkt Wesentlichen vom Unwesentlichen [und] Zufälligen [...]“ (aus „Meyers großes Taschenlexikon“, nach [Poe00, S. 105]) verstehen. Die Abstraktion erlaubt es den Modellnutzern, sich auf die interessanten Systemaspekte konzentrieren zu können. Zu möglichen Abstraktionen gehören die Ausschnittsbildung und die Klassifikation, d. h. die Identifikation von Typen. Auf letztere Form der Abstraktion werden wir detailliert in Kapitel 3 eingehen.

Im Falle eines Software-Systems kann es sich bei einem Modell, wie auch bei anderen Modellen aus dem ingenieurwissenschaftlichen Bereich, sowohl um eine *Beschreibung* eines existierenden Systems als auch um eine *Spezifikation* eines zu konstruierenden Systems handeln (siehe z. B. [Sei03]). In ersterem Fall nennt man solch ein Modell *deskriptiv*, im letzteren Fall *preskriptiv* [Lud03].

Ein Modell basiert auf einem *Formalismus*, der die *Syntax* (die Notation) und *Semantik* (die Bedeutung) der Modellelemente klar definiert. Die Syntax unterteilt sich dabei weiter in die *konkrete* und die *abstrakte Syntax*, bei der Semantik wird die *statische Semantik* („Well-formedness“-Regeln) und die *dynamische Semantik* unterschieden. Abb. 2-1 zeigt diese Zusammenhänge grafisch.

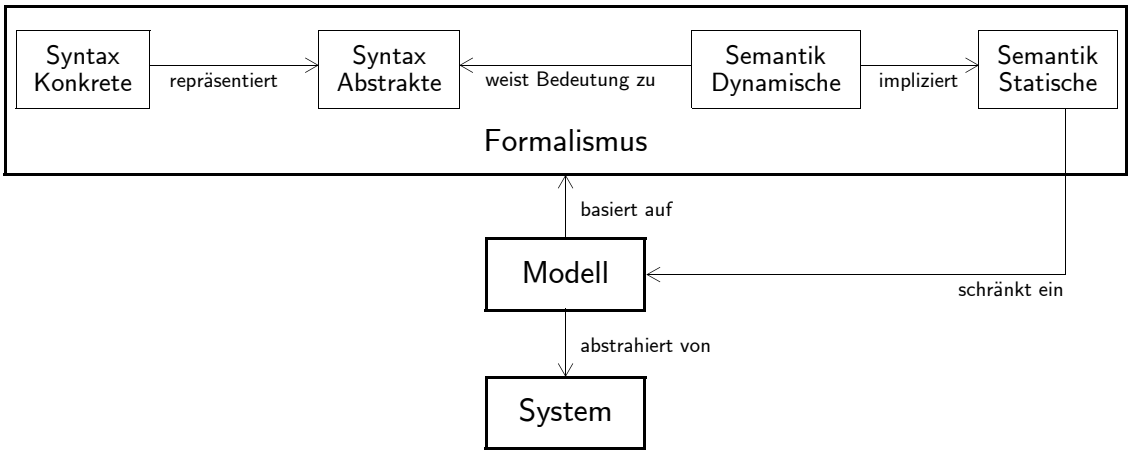


Abbildung 2-1. System, Modell und Formalismus (nach [CaS02])

Die Syntax eines Formalismus beschreibt *ausschließlich* dessen Notationsaspekte. Die Beschreibung der Bedeutung geschieht durch die Semantik [HaR00]. Die konkrete Syntax spezifiziert die lesbare Darstellung (oder Repräsentation) der abstrakten Notationselemente. So wird z. B. in dem Formalismus der Mathematik für das abstrakte Notationselement „Division“ das konkrete Element „ ÷ “ verwendet. Nehmen wir nun einem sehr kleinen Ausschnitt aus dieser Mathematik an, so können

wir weiter die Notationselemente „Zahl“ und „Ausdruck“ identifizieren, wobei letzterer z. B. die konkrete Form „$42 \div 2$ “ annehmen kann.

Zur Beschreibung der Bedeutung dieser abstrakten Notationselemente, wir wollen sie N nennen, bedarf es deren Abbildung A auf eine semantische Domäne D (siehe [HaR00]). Solch ein semantisches Mapping, welches den Notationselementen eine Bedeutung zuweist, lässt sich beispielsweise als Funktion

$$A : N \to D \qquad \textbf{(Gleichung 2-1)}$$

beschreiben. Wollen wir uns in dem Mathematikbeispiel z. B. auf die Division natürlicher Zahlen (Ganzzahldivision) beschränken, dann definieren wir die Abbildung

$$A_{\mathrm{m}} : N_{\mathrm{m}} \to \mathbb{N},$$

wobei $\mathbb{N}$ die Menge der natürlichen Zahlen (inkl. der Null) darstellt und N_{m} die abstrakten Notationselemente „Division“, „Zahl“ und „Ausdruck“ beinhaltet.

Die Bedeutungen der relevanten Notationselemente unseres Mathematikbeispiels können nun durch Abbildungen auf Elemente aus $\mathbb{N}$ festgelegt werden. Insbesondere kann man damit das semantische Mapping des Elements „Ausdruck“ induktiv definieren als:

$$A_{\mathrm{m}}(\text{Ausdruck}) \;=\; A_{\mathrm{m}}(\text{Zahl}) \div A_{\mathrm{m}}(\text{Zahl}),$$

wobei die „Zahlen“ in der üblichen Zahlendarstellung interpretiert werden, also z. B. $A_{\mathrm{m}}(42) \;=\; 42$.

Mit der Kenntnis der Bedeutung der Notationselemente ergeben sich nun unter Umständen auch Einschränkungen bzgl. der Verwendung dieser Elemente. Eine offensichtliche Einschränkung in obigem Beispiel ist, dass in einem gültigen Ausdruck nie die Zahl „0“ an der Stelle der zweiten „Zahl“ verwendet werden darf, da die Division durch Null nicht definiert ist. Auch wenn solche „Well-formedness“-Regeln in der Literatur (und auch in unserer Abbildung) als „statische Semantik“ bezeichnet werden, sei anzumerken, dass diese lediglich die Syntax einschränken und nicht die Bedeutung der Notationselemente klären [HaR00, S. 16].

2.1.1 Formalisierung und Automatisierung

Sowohl für die Spezifikation der Syntax als auch für die Beschreibung der Semantik kann man unterschiedliche Grade der Formalisierung einsetzen. Im obigen Mathematikbeispiel hatten wir für die Definition der dynamischen Semantik eine formale Beschreibung (vgl. [WiD94, S. 33]) angegeben. Auch für die Syntax lässt sich eine solche Formalisierung erreichen.

Der Formalisierungsgrad hat eine bedeutende Auswirkung auf den Grad der möglichen Automatisierung. Da jede Software-Entwicklungsaktivität eine Bearbeitung (Lesen oder Modifizieren) von Modellen darstellt, muss zur Automatisierung solcher Aktivitäten das zu bearbeitende Modell *maschinenlesbar* (und *-modifizierbar*) sein. Nur formale Aspekte eines Modells erfüllen die Voraussetzung für eine Maschinenlesbarkeit und sind daher einer Behandlung durch Software-Werkzeuge zugänglich [OdR99]. Abb. 2-2 zeigt die Konsequenzen für die Automatisierung in Abhängigkeit vom gewählten Formalisierungsgrad der Syntax und Semantik eines Formalismus.

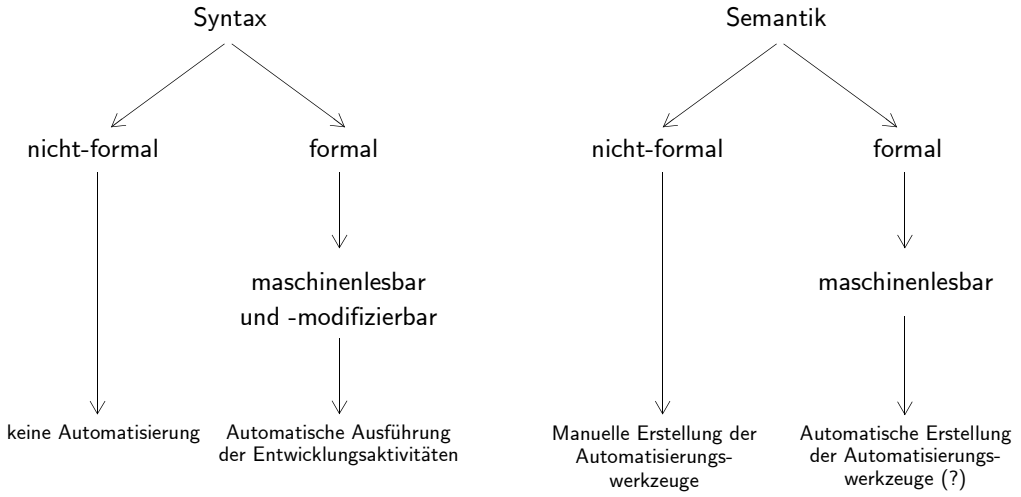


Abbildung 2-2. Automatisierungsgrad in Abhängigkeit vom Formalisierungsgrad

Die Minimalanforderung an den Formalismus zur Beschreibung von Modellen für eine Automatisierung ist, dass dessen Syntax formal ist. Nur dadurch lassen sich Änderungen an den Modellen überhaupt per Werkzeugeingriff vornehmen.

Solche Modelländerungen sind normalerweise syntaktischer Natur. Allerdings muss bei der Erstellung der Werkzeuge selbstverständlich ein klares Verständnis der Bedeutung der Modellelemente vorliegen, da sich sonst keine sinnvollen syntaktischen Transformationen formulieren ließen. So würde z. B. die Automatisierung der Aktivität der Vereinfachung unserer mathematischer Ausdrücke ohne die Kenntnis der Bedeutung der Division nicht möglich sein. Woher sollte man sonst wissen, dass man die zwei in einem Divisionsausdruck zu findenden Zahlen durch die Anwendung der mathematischen Operation der Division zu einer einzelnen Zahl reduzieren kann?

Eine theoretische, über die manuelle Erstellung der Werkzeuge hinausgehende Automatisierung wäre denkbar, wenn die Werkzeuge die formale Semantik „lesen“ und „verstehen“ könnten, weil dadurch auch eine automatische Erstellung der Automatisierungswerkzeuge möglich wäre. Desweiteren wäre sogar eine Modifikation der Semantik denkbar, deren Konsequenzen unserer Meinung nach aber nicht mehr

beherrschbar wären. Zudem gibt es i.d.R. keine Werkzeuge, die eine direkte Manipulation der Semantik ermöglichen [HaR00]. In dieser Arbeit wollen wir daher von einer natürlichsprachlichen und damit nicht-formalen Beschreibung der Semantik ausgehen. Dadurch wird eine manuelle Erstellung der Automatisierungswerkzeuge möglich, womit man bereits deutliche Qualitäts- und Effizienzgewinne erreichen kann (siehe Kapitel 10).

Für die Realisierung solcher Werkzeuge benötigen wir nun also eine formale Spezifikation der Syntax, welche sowohl die konkreten als auch die abstrakten Notationselemente beinhaltet. Daher werden wir in den folgenden Unterabschnitten zwei Techniken einer solchen Formalisierung, *Grammatiken* und *Metamodelle*, vorstellen und deren Eignung zur Beschreibung der Syntax diskutieren.

2.1.2 Grammatiken

Die Syntax des Formalismus, auf dem ein Modell basiert, kann man auch als *Sprache*, in der das Modell ausgedrückt wird, auffassen. Daher bieten sich Grammatiken, die per Definition einer „deklarativen und transparenten Definition von Sprachen dienen“ [SpH96, S.133], zur Formalisierung an.

Zunächst müssen wir allerdings klären, was ein Wort einer möglichen Sprache überhaupt sein kann. Dazu geben wir die bekannte Definition aus der theoretischen Informatik an [SpH96, S.6].

Definition 2-1: Ein *Alphabet* Σ ist eine nichtleere Menge von sogenannten *Buchstaben*. Ein **Wort** über Σ ist eine endliche Folge $a = a_1 \ldots a_n$ von Buchstaben aus Σ. Die Menge aller Wörter über Σ bezeichnen wir mit Σ^*.

Jetzt lässt sich eine Grammatik und die von einer Grammatik erzeugten Sprache [SpH96, S.133][HoU88, S.83] definieren:

Definition 2-2: Eine **Grammatik** G ist ein Tupel $G = (N, T, \Pi, Z)$. Dabei ist N eine endliche Menge von *Nichtterminalsymbolen*, T eine endliche Menge von *Terminalsymbolen* mit $N \cap T = \varnothing$ und Π eine endliche Menge von Produktionen der Form $l ::= r$ mit $l, r \in (N \cup T)^*$ und der Eigenschaft, dass l mindestens ein Zeichen aus N enthält. Schließlich ist Z ein Startsymbol mit $Z \in N$.

Definition 2-3: Die von einer Grammatik G erzeugte **Sprache** ist die Menge aller Wörter $w \in T^*$, die sich ausgehend vom Startsymbol Z durch die Anwendung der Produktionsregeln Π *ableiten* lassen.

Für die praktische Anwendung reicht oft eine Untermenge der Grammatiken, die *kontextfreien Grammatiken*, aus [HoU88, S.82]. Diese sind dadurch gekennzeichnet, dass für jede Produktion $l \in N$ gelten muss.

Eine bekannte Notation zur Beschreibung der Produktionsregeln einer kontextfreien Grammatik ist die *Backus-Naur-Form* (BNF, siehe [MaL87][HoU88]). Für unser Mathematikbeispiel aus Abschnitt 2.1 können wir z.B. die folgenden Produktionsregeln zur Beschreibung der konkreten Notationselemente angeben (die eingeklammerten Begriffe stellen die Nichtterminalsymbole und das Zeichen „| “ eine Alternative dar):

```
<Ausdruck>   ::= <Zahl> ÷ <Zahl>
<Zahl>       ::= <Zahl> | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 0
```

Als Startsymbol wählen wir Z = `<Ausdruck>`. Zahlen bestehen nach dieser Grammatik also aus einer beliebigen (nichtleeren) Folge von Ziffern (wobei führende Nullen der Einfachheit halber erlaubt sind). Ein Ausdruck besteht immer aus zwei Zahlen, die durch ein Divisionssymbol verknüpft sind.

Aus obigen Definitionen können wir nun folgern, dass jedes Modell, das in einem gegebenen Formalismus ausgedrückt wird, aus einem konkreten Wort besteht, das jeweils durch Anwendung der Produktionsregeln ausgehend von dem Startsymbol entstanden ist.

Will man computergestützt auf solch ein Modell zugreifen oder dieses modifizieren, bietet sich eine Repräsentation an, die von dem inhärent textuellen Charakter der Wörter abstrahiert. Die typische Darstellung, die man z.B. auch bei Programmiersprachen-Compilern findet ist die der *Ableitungs-* oder *Parsebäume* [ASU97, S.31ff.][HoU88, S.86].

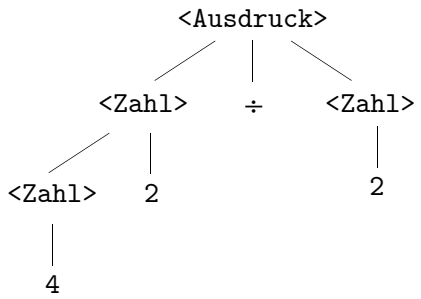


Abbildung 2-3. Beispielhafter Ableitungsbaum für mathematische Ausdrücke

Eine solche (grafische) Darstellung der Ableitungen gibt den einzelnen Wörtern eine Struktur, die bei einem computergestützten Zugriff auf die Wörter nützlich ist. In Abb. 2-3 ist solch ein Baum für das Beispiel `42 ÷ 2` zu sehen. Wie man erkennt sind die Knoten eines Ableitungsbaums entweder mit Terminal- oder Nichtterminalsymbolen markiert und das Startsymbol entspricht der Markierung der Wurzel.

Die bisher vorgestellte Art der Grammatik eignet sich sehr gut für die Beschreibung linearer Modelle (z.B. Programmcode), was mit der eindimensionalen Natur der Wörter der Sprache zu begründen ist. Für höherdimensionale Modelle (insbesondere zweidimensionale visuelle Modelle) ist eine solche Grammatik nur eingeschränkt sinnvoll, da man die höheren Dimensio-

nen auf eine einzelne Dimension zurückführen muss, was man typischerweise durch „Querverweise“ erreicht.

Ein Ansatz, der eine Beschreibung höherdimensionaler Modelle erlaubt, sind die *Graph-Grammatiken*, welche durch eine Erweiterung der obigen Grammatikdefinition beschrieben werden. Die „Wörter“ der durch eine Graph-Grammatik beschriebenen Sprache sind Graphen, welche in Anlehnung an [ReS97, S. 10] folgendermaßen definiert werden können:

Definition 2-4: Ein **Graph** G ist ein Tupel $G = (V, E, s, t)$. Dabei ist V eine endliche Menge von *Ecken* (Knoten, Punkten, „Vertices“) und E eine endliche Menge von *Kanten* (Bögen, „Edges“). Desweiteren sind $s: E \to V$ bzw. $t: E \to V$ Abbildungen, welche einer Kante eine Quelle bzw. ein Ziel zuweisen.

Eine Graph-Grammatik beschreibt nun die zulässigen Graphen für ein gegebenes Alphabet von Ecken und Kanten. In Anlehnung an [ReS97, S. 14 und S. 11] gilt:

Definition 2-5: Eine **Graph-Grammatik** GG ist ein Tupel (V, E, Π, A). Dabei ist V eine Menge von Ecken und E eine Menge von Kanten. Desweiteren ist A ein nichtleerer initialer Graph, welcher *Axiom* genannt wird, mit $A = (V_A, E_A, s_A, t_A)$, $V_A \subseteq V$ und $E_A \subseteq E$. Schließlich ist Π eine Menge von *Graph-Grammatik-Produktionen* der Form $L ::= R$, wobei L und R Graphen über V und E sind. Dabei wird L *als „Left-Hand Side“* und R als *„Right-Hand Side“* der Regel bezeichnet.

Konkret erfolgt eine Produktion in einer Graph-Grammatik durch das Finden von Instanzen der linken Seite einer Produktionsregel und das Ersetzen der gefundenen Teilgraphen durch die rechte Seite der Regel [AEH99]. Auf eine analoge Form der Graphersetzung werden wir dabei bei der Beschreibung von Graphtransformationen in Abschnitt 4.2 auf Seite 61 nochmals zurrückkommen.

Wie auch bei den linearen Grammatiken ist die von einer Graph-Grammatik GG erzeugte Sprache die Menge aller Graphen G, die sich ausgehend vom Axiom A durch Anwendung der Produktionsregeln aus Π ableiten lassen.

An dieser Stelle wollen wir ein kleines Beispiel einer Graph-Grammatik geben, weil wir diese Technik im weiteren Verlauf der Arbeit nutzen werden. Als Beispiel wählen wir hierbei die Darstellung von Binärbäumen. Die Notation der in Abb. 2-4 dargestellten Graph-Grammatik orientiert sich dabei an der in [BTM99].

Das Axiom ist binärBaum. Nichtterminalsymbole sind durch gerundete Rechtecke dargestellt. Lässt sich der Inhalt eines Symbols durch eine lineare Grammatik beschreiben, so ist dies durch <nichtTerminal> dargestellt. So erfolgt in obigem Beispiel die Markierung der Knoten des Binärbaums durch eine Zahl, wie sie in der linearen Grammatik der Mathematikausdrücke definiert wurde.

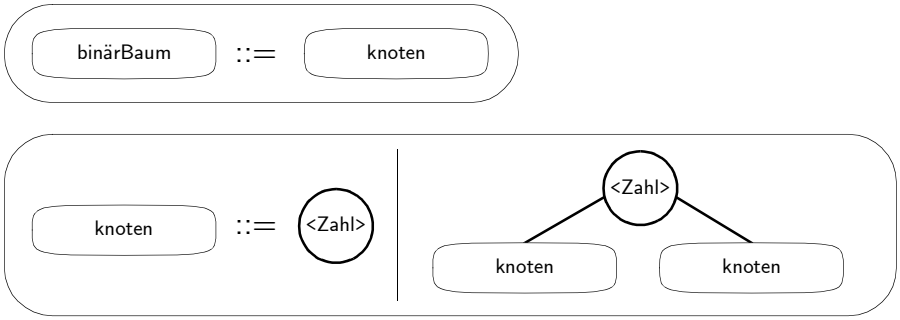


Abbildung 2-4. Graph-Grammatik zur Beschreibung von Binärbäumen

Auch wenn das obige Beispiel durch eine kontextfreie Graph-Grammatik definiert wurde (L beinhaltete jeweils nur ein Nichtterminalsymbol), ist dies i.d.R. zu restriktiv für die Definition komplexer visueller Sprachen (siehe [ReS97, S.4]). Daher werden dafür kontextsensitive Grammatiken eingesetzt. Definition 2-5 erlaubt beide Varianten. Anders als bei linearen Grammatiken ist das Ersetzen von Nichtterminalen bei Graph-Grammatiken allerdings nicht so klar. Eine Möglichkeit ist das Einführen von Kontextelementen, die sowohl in L als auch in R auftauchen und somit die Verbindung neuer Teile zu den alten Teilen erlaubt. Für weitere Details sei auf die Veröffentlichung von Rekers und Schürr [ReS97] verwiesen.

Wie wir an Hand der obigen Beispiele bereits gesehen hatten, sind Grammatiken eine sehr gute Wahl zur Beschreibung der konkreten Syntax von Modellen. Für die Definition der konkreten Syntax linearer Modelle, wie z.B. Programmcode, eignen sich die kontextfreien (linearen) Grammatiken. Für die Spezifikation der konkreten Syntax höherdimensionaler (visueller) Modelle bieten sich die Graph-Grammatiken an. Die besondere Eignung der Grammatiken zur Beschreibung der konkreten Notationselemente liegt darin begründet, dass die Grammatiken explizit die Nachbarschaft (oder Topologie) der Syntaxelemente beschreiben. So ist bei unserem Mathematikbeispiel durch die Grammatik exakt beschrieben, welche Zahl vor und welche Zahl nach dem Divisionssymbol zu stehen hat; bei der Binärbaumgrammatik ist explizit definiert, welcher Kindknoten links und welcher rechts unter einem Elternknoten hängt.

Grammatiken werden schon seit langem eingesetzt, weshalb sich eine große Auswahl von Werkzeugen für den Umgang mit grammatikbasierten Modellen bietet. Als Beispiel sei hier der bereits in den 1970er Jahren entwickelte Parser-Generator Yacc genannt, der aus einer gegebenen Grammatik automatisch C-Code erzeugt [ASU97, S.313ff.]. Moderne Parser-Generatoren, wie z.B. ANTLR [PaQ95], erzeugen sogar Code zum Aufbau und zum Zugriff auf Ableitungsbäume. Die Einsatzbereiche solcher Werkzeuge sind traditionell Hochsprachen-Compiler.

Zur Beschreibung der abstrakten Notationselemente sind die durch Grammatiken explizit beschriebenen Topologie-Informationen allerdings nicht notwendig. In

einem Binärbaum ist es zum Beispiel nicht relevant, ob die Kinderknoten links oder rechts stehen. In einem mathematischen Ausdruck genügt es zu wissen welche Zahl der Dividend und welche Zahl der Divisor ist. Insbesondere sieht man an dem Mathematikbeispiel, dass die „Reihenfolge“ der Zahlen austauschbar ist, denn es gilt

$$42 \div 2 \;=\; \frac{42}{2} \;=\; 2^{-1} \cdot 42\,.$$

Natürlich lässt sich auch die abstrakte Syntax mit Grammatiken beschreiben. So existiert z.B. eine Abstraktion der Parsebäumen, die *abstrakten Syntaxbäume* (oder einfach *Syntaxbäume*), die ihrem Namen nach auf der abstrakten Syntax eines Formalismus basieren [ASU97, S.60f.]. Da man allerdings in solchen abstrakten Modellen oftmals nicht von der expliziten Beschreibung von Nachbarschaften Gebrauch macht, bietet sich an dieser Stelle eine andere Technik, die Metamodellierung, an, welche auf die zum Teil aufwändige Spezifikation von Produktionsregeln verzichtet.

2.1.3 Metamodelle

Kern der Metamodellierung ist die Erkenntnis, dass ein Modell (oder genauer dessen Formalismus, vgl. [CaS02]) auch wieder als System aufgefasst werden kann, also als ein spezielles System. Damit kann man nun den Formalismus auch wieder durch ein Modell beschreiben, welches folglicherweise Metamodell (von „meta“, griech. „nach“) genannt wird. Abbildung 2-5 zeigt diesen Sachverhalt zur Illustration nochmals grafisch.

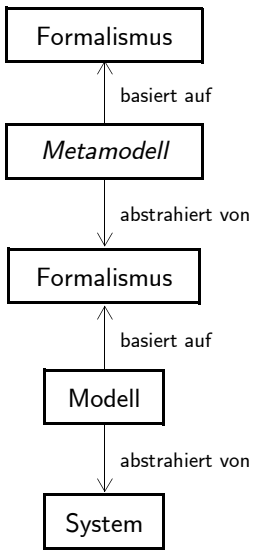


Abbildung 2-5. Interpretation eines Metamodells

Die Wahl des Formalismus, auf dem das Metamodell basiert, ist dabei prinzipiell freigestellt. Typischerweise verwendet man aber eine objektorientierte Begriffswelt (eine konkrete Syntax einer solchen OO-Modellierungssprache ist die bekannte UML [RJB99]). Desweiteren ist es sinnvoll, die einzelnen Formalismen (des Modells und des Metamodells) so zu wählen, dass ähnliche (oder gleiche) Konzepte verwendet werden, was die Allgemeingültigkeit und das Verständnis der (Meta-)modelle erhöht (diese Verwendung könnte man als „echte“ Metamodellierung bezeichnen, siehe z.B. [Ken02, S.288]).

Genauer werden wir in Kapitel 3 auf den angesprochenen OO-Formalismus zur Spezifikation von Modellen eingehen. Zur Illustration des Metamodellansatzes wollen wir in Abb. 2-6 das kleine Mathematikbeispiel mit Hilfe der UML-Klassen- und Objektdiagramme vorstellen und setzen an dieser Stelle Grundkenntnisse dieser Notation voraus (siehe z.B. [RJB99, S.41ff.]).

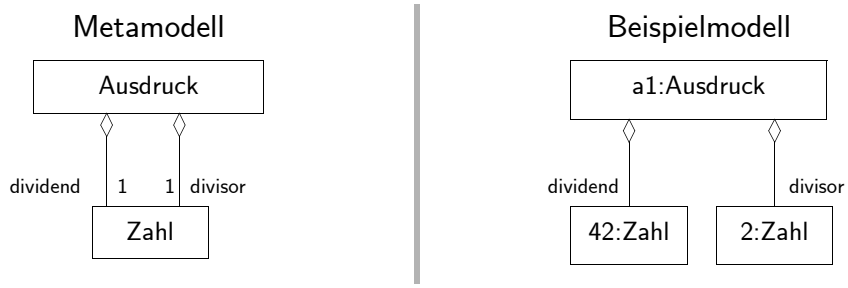


Abbildung 2-6. Metamodell für mathematische Ausdrücke und Beispielmodell

Das abstrakte Notationselement „Ausdruck“ wird in unserem Metamodell durch einen Objekttyp Ausdruck beschrieben. Jede Instanz eines solchen Objekttyps aggregiert genau zwei Instanzen des Objekttyps Zahl. Durch die Benennung der Beziehungsenden (dividend und divisor) wird darüberhinaus den Zahlen eine eindeutige Rolle in einem Ausdruck zugewiesen. Damit könnten wir z.B. ganz einfach die obige Forderung nach einem Divisor größer Null umsetzen, indem wir verbieten, dass eine Zahl-Instanz in der Rolle des Divisors einem Wert gleich Null entspricht.

Die einer Grammatik zu Grunde liegenden Konzepte (Wörter, Produktionsregeln, etc.) sind sehr wenige und lassen sich knapp und präzise definieren, womit sich Grammatiken eindeutig anwenden und verstehen lassen. Bei der Metamodellierung hingegen wird die Menge dieser Konzepte, insbesondere bei einer objektorientierten Begriffswelt, schon viel größer. Eine einheitliche Begriffsdefinition hat sich bisher leider nicht vollständig etabliert (siehe dazu Abschnitt 3.2.3 auf Seite 51). Wir wollen in dieser Arbeit aber versuchen einen sinnvollen Kern von Begriffen zu definieren (siehe Abschnitt 3.2 auf Seite 38), der als Grundwortschatz für den Rest der Arbeit dient.

2.2 Explizite Modelle

Nachdem wir im vorangegengenen Abschnitt auf mögliche Formalismen von Modellen eingegangen sind, sollen in diesem Abschnitt wichtige Arten von Modellen in der Software-Entwicklung genauer betrachtet werden. Am Anfang dieses Kapitels hatten wir ja bereits auf die beiden wichtigsten Modellarten hingewiesen: das Produkt- und das Prozessmodell. In den folgenden Abschnitten wollen wir auf diese beiden Arten detaillierter eingehen und deren Beitrag für eine Automatisierung herausstellen.

2.2.1 Produktmodell

Ein Produktmodell stellt eine Abstraktion von den Entwicklungsprodukten (oder *Artefakten*) und den Beziehungen zwischen diesen Artefakten dar [MeQ01]. Typischerweise handelt es sich bei den Entwicklungsprodukten in der Software-Entwicklung um Modelle (wie z.B. UML Diagramme [RJB99], Petri-Netze [Rei90] oder SDL-Spezifikationen [OFM94]).

Ein Produkt wäre z.B. das Anforderungsdokument für eine Heizungssteuerung, das aus den Benutzeranforderungen „in einem Raum soll eine angenehme Temperatur herrschen“ und „der Energieverbrauch soll minimiert werden“ besteht. Das zugehörige Produktmodell würde dann allgemeingültig beschreiben, dass alle berücksichtigten Anforderungsdokumente aus einer Menge von Benutzeranforderungen bestehen.

Ein Produkt kann man also als eine Instanz des zugehörigen Produktmodells verstehen. Eine solche Instanz eines Produktmodells beschreibt quasi einen „Schnappschuss“ der Artefakte während der Software-Entwicklung. Der letzte „Schnappschuss“ beinhaltet dann auch das endgültige (oder fertige) Produkt.

Ein Produktmodell erlaubt damit die statische Beschreibung von Entwicklungsprodukten durch die Spezifikation der statischen Struktur der Artefakte und deren Beziehungen zueinander.

In Kapitel 6 werden wir ein vollständiges Produktmodell einer Software-Entwicklungsmethode für reaktive Systeme vorstellen.

2.2.2 Prozessmodell

Ein Prozessmodell beschreibt die Aktivitäten in einem Entwicklungsprozess und die Abhängigkeiten zwischen diesen Entwicklungsaktivitäten [SPH02][MeQ01].

Eine Entwicklungsaktivität hat dabei eine Produktmodellinstanz (oder eine Teilmenge der Artefakte) als Eingabe und die durch die Aktivität erzeugte Produktmodellinstanz als Ausgabe. Ein Prozessmodell basiert also immer auf den im Produktmodell beschriebenen Artefakten. Ein Beispiel für eine Aktivität wäre z.B. die Inspektion der Anforderungsdokumente auf Inkonsistenzen. Eingabe wäre ein konkretes Dokument (z.B. unser obiges Anforderungsdokument für die Heizungssteuerung) und die Ausgabe wäre eine Fehlerliste, welche die aufgedeckten Inkonsistenzen beinhaltet.

Die Abhängigkeiten zwischen den Aktivitäten definieren einen *möglichen* Ablauf dieser Aktivitäten während des Entwicklungsprozesses [SPH02]. Zum Beispiel muss zunächst ein Anforderungsdokument existieren bevor man mit der Spezifikation von Systemkomponenten beginnen kann. Eine solche Abfolge von Entwicklungsaktivi-

täten impliziert damit auch eine Sequenz von „Schnappschüssen“ von Artefakten, also Produktmodellinstanzen.

In Abb. 2-7 sind die Rollen von Produkt- und Prozessmodell nochmals illustriert.

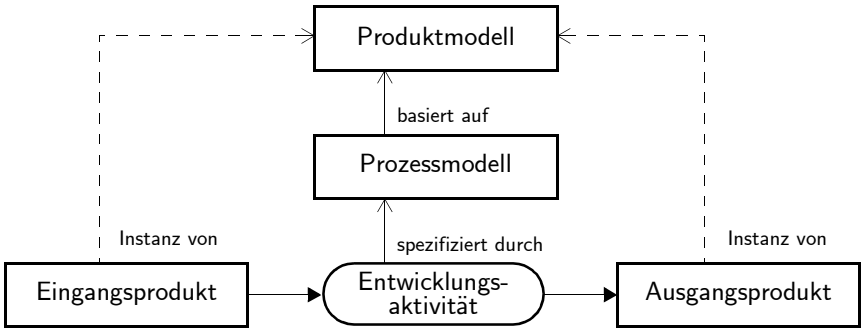


Abbildung 2-7. Rolle von Produkt- und Prozessmodell zur Beschreibung einer Aktivität

In dieser Abbildung ist eine Notation verwendet, die wir im Verlauf dieser Arbeit für Prozessmodelle (oder zumindest Ausschnitte aus diesen Modellen) verwenden werden. Dabei bezeichnen die Rechtecke die Produkte (Artefakte) und die abgerundeten Elemente die Aktivitäten. Ein Pfeil mit einer gefüllten Pfeilspitze, der an einer Aktivität endet, beschreibt eine Eingabeabhängigkeit. Endet ein solcher Pfeil bei einem Produkt, dann handelt es sich um eine Ausgabeabhängigkeit. Die sonstigen Pfeiltypen beschreiben andere Beziehungen.

Es sei an dieser Stelle erwähnt, dass sowohl die einzelne Aktivität (also die Abbildung zwischen Eingangs- und Ausgangsprodukt) also auch die Ausführung der Aktivitäten unter Berücksichtigung deren Abhängigkeiten automatisiert werden kann. Der Hauptteil dieser Arbeit liegt dabei auf der Automatisierung einzelner Aktivitäten. Auf der Basis solcher automatischer Aktivitäten wird aber dann auch eine neue Art der Automatisierung der Prozessdurchführung möglich, welche prinzipiell ohne Benutzerintervention auskommen könnte. In Abschnitt 9.3 auf Seite 275 werden wir dazu Ergebnisse präsentieren.

Klassifikation von Aktivitäten

In diesem Abschnitt wollen wir nun – nach der obigen allgemeinen Einführung von Entwicklungsaktivitäten – verschiedene Ausprägungen solcher Aktivitäten genauer beleuchten. Dies erlaubt uns dann im Verlauf dieser Arbeit die besprochenen Aktivitäten präzise einordnen zu können.

Wir hatten bereits erwähnt, dass es sich bei Software-Entwicklungsartefakten i.d.R. um Modelle handelt (auch Code kann man in diesem Sinne als Modell des lauffähigen Systems betrachten). Daher handelt es sich bei Software-Entwicklungsaktivitäten im allgemeinsten Sinne um *Modelltransformationen*.

Da wir bereits auf die Rolle von Modellen als Spezifikationen von Software-Artefakten hingewiesen haben, wollen wir an dieser Stelle auch auf die Abhängigkeiten zwischen Modell und modelliertem System eingehen. In Anlehnung an eine Schreibweise von Caplat und Sourrouille [CaS02] sei nun M das Modell eines Systems S und F der Formalismus, in welchem das Modell beschrieben wird. Jede Abbildung zwischen Modellen (mit zugehörigem Formalismus) lässt sich dann formulieren als

$$M_1(S_1)\big|_{F_1} \rightarrow M_2(S_2)\big|_{F_2}. \qquad \textbf{(Gleichung 2-2)}$$

Dabei stellt M_1 das Eingangsmodell und M_2 das Ausgangsmodell der Abbildung dar. Eine solche Abbildung sagt dabei nichts darüber aus, ob dabei das Eingangsmodell konsumiert oder modifiziert wird; dies ist orthogonal zu der hier beschriebenen Abbildung zu sehen. Desweiteren kann eine Transformation (und damit eine Entwicklungsaktivität) *monolithisch* (oder *atomar*) sein oder aus mehreren einzelnen Schritten bestehen [HHS02][Que02, S.115f.] und damit mehrstufig erfolgen. Caplat und Sourouille [CaS03] unterscheiden Modelltransformationen weiter in *endogene Transformationen* ($F_1 = F_2$) und *exogene Transformationen* ($F_1 \neq F_2$).

Abhängig davon, ob Modelle, Formalismen oder Systeme gleich oder unterschiedlich sind, wollen wir ein Klassifikationsschema der Transformationen einführen, welches feingranularer als die bisher in der Literatur vorgestellten Schemata ist. Abb. 2-8 zeigt dazu zunächst eine Übersicht in Form eines „Entscheidungsbaums“.

Transformation vom Typ S0

Dass sich, obwohl das Modell sich nicht geändert hat, eine Änderung des Systems ergeben haben soll (Typ S0) klingt zunächst absurd. Bedenkt man allerdings, dass ein Modell immer nur eine Abstraktion des Systems darstellt und daher nicht alle Systemeigenschaften reflektiert, dann wird ein solcher Fall plausibel. Immer dann, wenn sich Systemeigenschaften, die nicht im Modell beschrieben wurden, ändern, dann ändert sich auch das System ohne dass ein Modelländerung nötig wäre. Auch Queins [Que02, S.177ff.] weist auf einen ähnlichen Sachverhalt bei der Beziehung zwischen Spezifikation (Modell) und Prototyp (ausführbares System) hin.

Ein anschauliches Beispiel für eine Systemeigenschaft, die nicht in einem Modell beschrieben ist, liefern die Petri-Netze (siehe z.B. [Rei90]). Ein Petri-Netz ist allgemein ein gerichteter bipartiter Graph, der zwei unterschiedliche Typen von Knoten aufweist: Plätze und Transitionen. Dabei können die Plätze mit sog. Token markiert werden. Eine Feuerregel beschreibt, wann Transitionen ausgelöst werden *können*. I.d.R. ist dies der Fall, wenn alle Plätze, die mit dem Eingang einer Transition verbunden sind mindestens mit einem Token markiert sind. Petri-Netze benutzt man sehr häufig um Prozessabläufe abstrakt zu beschreiben. Dabei verwendet man die Transitionen zur Beschreibung von Ereignissen und die Plätze zur Modellierung von

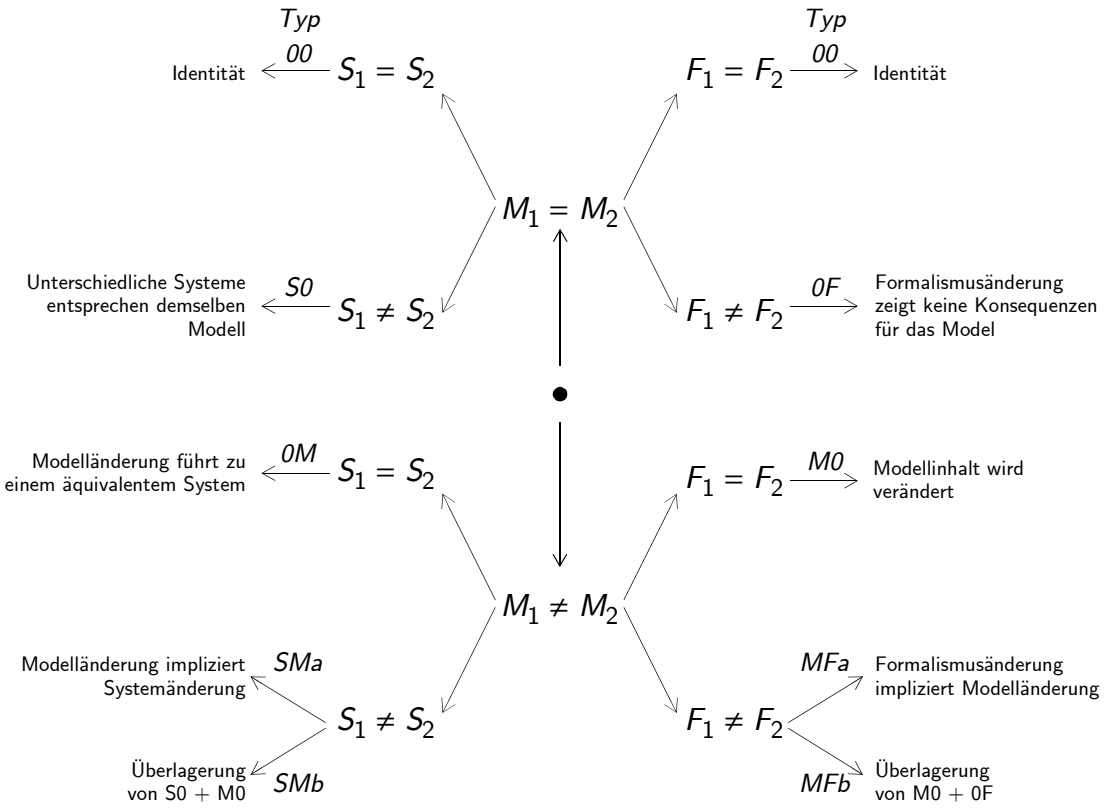


Abbildung 2-8. Verschiedene Typen der Modelltransformation („Entscheidungsbaum“)

Vor- und Nachbedingungen. Da für jede Transition in einem Petri-Netz allerdings immer nur ein *potenzielles* Feuern beschrieben ist, wird durch das gesamte Netz nur *mögliche* Abfolgen von Prozessereignissen beschrieben. Daher wird jedes reale System, das eine dieser möglichen Abfolgen aufweist, korrekt durch das (abstraktere) Petri-Netz-Modell beschrieben.

Transformation vom Typ 0M

Manche Modelländerungen liefern ein bzgl. des modellierten Systems äquivalentes Modell (Typ 0M). So ist es offensichtlich, dass man z. B. trotz der Änderung eines Variablennamens `i` zu `zaehlVariable` letztlich ein identisches Software-System spezifiziert.

Transformation vom Typ SM

Eine Abbildung vom Typ SM besitzt zwei Facetten. Zunächst handelt es sich um den offensichtlichen Fall, dass eine Modelländerung auch eine Änderung des spezifizierten Systems impliziert (Typ SMa). Ergänzt man z. B. ein Software-System um eine neue Komponente, so wird sich das erweiterte System sicherlich von dem Anfangssystem unterscheiden.

Die beobachtete Änderung von Modell und System kann aber auch durch eine „Überlagerung“ entstanden sein (Typ SMb). Dies ist theoretisch dann möglich, wenn die Modelländerung keine Systemänderung impliziert (siehe Typ 0M), das System aber dennoch Eigenschaften besitzt, die nicht im Modell beschrieben wurden (siehe Typ S0).

Transformation vom Typ 0F

Die Transformation vom Typ 0F führt lediglich zu einer Änderung der Darstellung des Modells, d.h. die konkrete Syntax von Formalismus F_1 unterscheidet sich von der von Formalismus F_2. Eine Änderung der abstrakten Syntax oder gar der Semantik hätte eine Modelländerung zur Folge (siehe Typ MF).

Ein Beispiel für eine solche Transformation liefert die Modellierungssprache SDL (siehe [ITU99] und [OFM94]), die neben einer grafischen Notation (SDL-GR) auch eine textuelle Repräsentation besitzt (SDL-PR, siehe dazu [EHS97, S.14f.]). In Abb. 2-9 sind beide Darstellungen eines SDL-Modells gezeigt. Das Modell beschreibt einen Zustandsübergang von Zustand Z1 nach Zustand Z2 bei Empfang eines Signals S1.

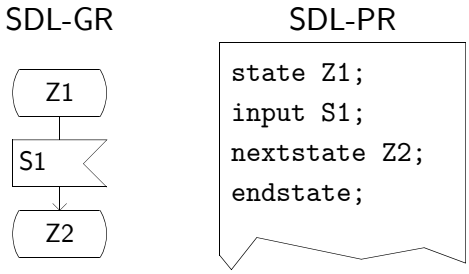


Abbildung 2-9. Unterschiedliche Repräsentationen desselben Modells

Transformation vom Typ M0

Transformationen vom Typ M0 sind „echte“ Modelltransformationen, d.h. die Modellinformation wird (bei Beibehalten des Formalismus) geändert. Das oben bereits angeführte Beispiel des Hinzufügens einer Komponente fällt in diese Klasse.

Caplat und Sourouille [CaS02][CaS03] nennen solch eine Modelltransformation auch *direkt*. Queins [Que02, S.117ff.] klassifiziert solche Transformationen weiter nach dem Ergebnis der jeweiligen Aktivität. So schlägt er die Klassen „*create*“, „*extend*“, „*refine*“, „*change*“, „*correct*“, „*test*“ und „*review*“ vor, welche die Erzeugung einer neuen Modellinformation, die Modifikation eines existierenden Modellelements bzw. das Testen oder die Inspektion eines existierenden Elements (mit der Erzeugung einer neuen Version des inspizierten Elements) repräsentieren.

Transformation vom Typ MF

Bei dem Typ MF, der letzten Art von Transformation, lassen sich analog zu dem Typ SM wiederum zwei Facetten identifizieren. Die offensichtlichste ist, dass eine Änderung des Formalismus eine Modelländerung impliziert (Typ MFa). Dies passiert immer dann, wenn sich die abstrakte Syntax oder die Semantik des Formalis-

mus ändert. Wechselt man z.B. vom Formalismus der UML-Objektdiagramme (wo Objekte standardmäßig abhängige Kontrollflüsse besitzen) zu dem Formalismus der Modellierungssprache SDL (wo Objekte als Prozesse voneinander entkoppelt sind), dann hat dies weitreichende Konsequenzen für das Modell und (bei mehr als einem Objekt) konsequenterweise auch für das modellierte System.

In der Model Driven Architecture (MDA, siehe Abschnitt 5.3.2 auf Seite 120) taucht diese Form der Transformation z.B. dann auf, wenn eine Abbildung von einem Platform Independent Model (PIM) auf ein Platform Specific Model (PSM) stattfindet, weil sich die Bedeutung der Modellelemente zwischen diesen beiden Modellen unterscheiden kann. So erfährt ein abstraktes Objekt in einem PIM z.B. die Abbildung auf einen Eintrag in einer Datenbank (im PSM).

Die zweite Facette (Typ MFb) beschreibt wiederum eine „Überlagerung“, d.h. dass obwohl eine (Syntax-)änderung des Formalismus keine Auswirkung hat (siehe Typ 0F) findet zusätzlich eine echte Modelländerung (wie bei Typ M0 beschrieben) statt.

Anwendung der Abbildungen

Zur effizienten Automatisierung von Entwicklungsaktivitäten muss man die Entwicklungsmodelle zunächst in eine geeignete interne Repräsentation für ein Werkzeug überführen. Das heißt, dass ein Eingangsmodell M_{e}, das in einer konkreten Syntax vorliegt, zunächst einmal in ein Modell $M_{\mathrm{e,\,i}}$ transformiert werden sollte, das von einer konkreten Syntax des Eingabemodells abstrahiert, denn ein solches „Ausblenden“ von konkreten Notationsaspekten vereinfacht eine Modelltransformation in einem erheblichen Maße.

Nach der erfolgten Transformation des abstrakten Modells $M_{\mathrm{e,\,i}}$, welche ein Modell $M_{\mathrm{a,\,i}}$ liefert, wird letzteres Modell dann auf ein Modell M_{a} in der gewünschten konkreten Syntax der Ausgangsmodelle abgebildet.

Zusammengefasst wird also die folgende Kette von Transformationen durchgeführt:

$$M_{\mathrm{e}} \xrightarrow{\mathrm{0F}} M_{\mathrm{e,\,i}} \xrightarrow{\mathrm{M0}} M_{\mathrm{a,\,i}} \xrightarrow{\mathrm{0F}} M_{\mathrm{a}}$$

Ein typisches Beispiel einer solchen zusammengesetzten Transformation sind Hochsprachen-Compiler. Eingangsmodell M_{e} ist der Quellcode des Programms, das es in Maschinensprache M_{a} zu überführen gilt. Dazu wird zunächst durch Parsen des Quellcodes ein (abstrakter) Syntaxbaum $M_{\mathrm{e,\,i}}$ aufgebaut (siehe Abschnitt 2.1.1 auf Seite 11). Dieser Baum, der als interne Datenstruktur des Compilers vorliegt, wird dann in den Syntaxbaum $M_{\mathrm{a,\,i}}$ des Ausgangsmodells transformiert. Aus diesem Baum erhält man dann durch einen „Unparse“-Vorgang den gewünschten Maschinencode. (Oftmals kürzt man den Weg auch ab und erzeugt das Ausgangsmodell

direkt aus dem Parsebaum oder Syntaxbaum des Eingangsmodells [ASU97, S.46ff.].)

Wie für obiges Compiler-Beispiel lassen sich mit Hilfe der vorgestellten Klassifikation viele andere Software-Entwicklungsaktivitäten hinsichtlich ihrer Zugehörigkeit zu einem Typ von Modelltransformation einordnen. In Tabelle 2-1 sind dazu einige Beispiele aufgelistet.

Tabelle 2-1. Klassifikation von beispielhaften Entwicklungsaktivitäten

Aktivität	**Typ der Transformation**
Hinzufügen einer neuen Komponente K: S_1 = System ohne K, S_2 = System mit K M_1 = Modell ohne K, M_2 = Modell mit K $F_1 = F_2$	 SMa M0
Berechnung der Systemkomplexität (z.B. Lines of Code): S_1 = analysiertes System, S_2 = Ergebnis M_1 = analysiertes Modell, M_2 = Ergebnismodell F_1 = C++, F_2 = natürliche Zahlen	 SMa MFa
Persistente (standardisierte) Speicherung eines UML-Modells: $S_1 = S_2$ $M_1 = M_2$ F_1 = Datenmodell des Werkzeuges, F_2 = XMI	 00 0F
Inspektion von Anforderungsdokumenten: S_1 = Anforderungen, S_2 = Inkonsistenzen M_1 = Anforderungsdokument, M_2 = Fehlerliste F_1 = Use Cases, F_2 = Text	 SMa MFa

Wie man erkennt, werden sowohl die „Berechnung der Systemkomplexität“ als auch die „Inspektion von Anforderungsdokumenten“ gleich klassifiziert. Man könnte auch sagen, dass beide Aktivitäten einer *Modellreduktion* (oder -*analyse*) entsprechen, da die Informationen des Eingabemodells auf wenige Daten im Ausgabemodell zurückgeführt wurden.

Eine weitere Gruppierung von Transformationen lässt sich nach dem Zweck der Abbildung vornehmen. So unterscheiden Czarnecki und Eisenecker in [CzE00, S.335f.] Transformationen zum Zwecke der *Evolution* (horizontale Transformation) und der *Implementierung* (vertikale Transformation).

Die Implementierungstransformation beschreibt dabei eine Abbildung des Eingangsmodells in ein Ausgangsmodell in einem Formalismus, der sich direkt auf einem Rechner (bzw. einer virtuellen Maschine) ausführen lässt. Eine Automatisierung solcher Transformationen ist weit verbreitet und begann mit der Einführung höherer Programmiersprachen, welche per Compiler in ausführbaren Code überführt werden. Mittlerweile ist eine solche Code-Generierung auch ausgehend von den abstrakteren Modellierungssprachen, wie z.B. SDL (siehe z.B. [OFM94]) oder auch der neuen UML 2 (siehe z.B. [BHK04][JRH03]) möglich. Insbesondere die letz-

te Form der Code-Generierung ist ein wichtiger Baustein der MDA, da dies die automatische Abbildung der Platform Specific Models auf Code bzw. Entwurfsmodelle erfordert (siehe z.B. [MiM03], [Ken02] und Abschnitt 5.3.2 auf Seite 120).

Die Automatisierung einer Evolutionstransformation ist dabei nicht so weit verbreitet. Dies liegt an der Tatsache, dass Programmcode (auch der einer höheren Programmiersprache) semantisch nicht reich genug für eine solche automatische Transformation ist. Erst mit der Einführung von Modellierungssprachen und darüberhinaus mit der Spezifikation spezifischer Produktmodelle für einzelne Anwendungsbereiche oder -methoden kann solch ein Vorhaben gelingen.

Czarnecki und Eisenecker postulieren sogar, dass der Umfang einer automatisierten Unterstützung mit der Zunahme der abstrakten Entwicklungsinformation, die in einem Eingabemodell vorhanden ist, wachsen wird [CzE00, S.338]. Ein wichtiger Beitrag unserer Arbeit ist genau solche Transformationen zu ermöglichen und dies an Hand konkreter Aktivitäten und Fallstudien auch zu belegen.

2.2.3 Weitere Modellarten

Durch die in einem Prozessmodell beschriebenen Aktivitäten und deren Abhängigkeiten über z.B. gemeinsame Produkte, wird nur eine *mögliche* Abfolge von Aktivitäten spezifiziert. Anders formuliert beschreibt ein Prozessmodell „wann eine Entwicklungsentscheidung potenziell getroffen werden *kann*“ [Que02, S.145]. Schränkt man diese potenzielle Aktivitätsausführung weiter ein und spezifiziert, „wann eine Entwicklungsentscheidung getroffen werden *soll*“ [Que02, S.145], erhält man ein sogenanntes *Vorgehensmodell*. Ein solches Vorgehensmodell ist insbesondere bei der Planung von Entwicklungsprojekten relevant, da Faktoren wie Arbeitszeiten und verfügbare Entwickler den Zeitpunkt von Entwicklungsentscheidungen beeinflussen. Siehe dazu zum Beispiel die Arbeiten von Rombach [Rom03] und Goldmann et al. [GHR03].

In dieser Arbeit wollen wir uns mit der Automatisierung solcher Planungs- oder Projektmanagementaufgaben nicht ausführlich befassen und werden daher auch Modelle, die in diesem Zusammenhang stehen (wie z.B. Modelle zur Fehlerabschätzung), nicht detaillierter einführen. Kurz werden wir dieses Gebiet allerdings in Abschnitt 9.3 auf Seite 275 anschneiden, wo wir – wie bereits erwähnt – eine automatisierte Ausführung von automatisierten Aktivitäten betrachten.

Zusammenfassung

Die Rolle von Modellen in der Software-Entwicklung wurde in diesem Kapitel beleuchtet und speziell der Einfluss von Syntax und Semantik, bzw. deren Formalisierungsgrad, auf eine Automatisierung untersucht.

Die zwei wichtigsten Vertreter von Modellen in der Software-Entwicklung, die Produkt- und Prozessmodelle wurden eingeführt und deren unterschiedliche Rollen in einem Entwicklungsprozess klargestellt.

Schließlich wurde eine sehr feine Klassifikation von Modellabbildungen (oder Modelltransformationen) vorgestellt, welche im weiteren Verlauf der Arbeit die präzise Einordnung und damit Beurteilung der betrachteten Modelltransformationen ermöglicht. Insbesondere wurde dabei auf die Rolle der Abbildung von einem Modell in einer konkreten Syntax auf ein Modell in einem abstrakteren Formalismus und umgekehrt hingewiesen. Durch eine vor- bzw. nachgeschaltete Abbildung dieses Typs lässt sich die eigentliche Modelländerung sehr einfach auf einem abstrakten Modell beschreiben.

Im folgenden Kapitel werden wir genauer auf eine der vorgestellten Techniken zur Formalisierung der Syntax eingehen: die Metamodellierung.

3 Grundlagen der objektorientierten Modellierung

The theory of logical types [...] *recommended itself to us in the first instance by its ability to solve certain contradictions* [...]. *But the theory in question is not wholly dependent upon this indirect recommendation: it has also a certain consonance with common sense which makes it interestingly credible.*

— Alfred North Whitehead, Bertrand Russell
[Principia Mathematica, 1913]

Die objektorientierte Modellierung und insbesondere die Metamodellierung ist eine wichtige Grundlage für den in dieser Arbeit vorgestellten Automatisierungsansatz, da die zu transformierenden Software-Spezifikationen und die Transformationen selbst mit dieser Technik beschrieben werden. Daher wird in diesem Kapitel zunächst genauer auf die Grundlagen der Modellierung und Metamodellierung eingegangen. Dazu gehört die Vorstellung möglicher Modellkonzepte, und -ebenen aber auch die Diskussion eventueller Probleme.

Obwohl für die Modellierung von Software-Systemen meist ein ungefähres Verständnis der Modellierungskonzepte ausreicht, muss man bei der Definition von Metamodellen und zugehörigen Werkzeugen (wie z.B. unseren Automatisierungs-Tools) ein präzises Verständnis der verwendeten Konzepte besitzen (siehe auch [OdR99]). Da noch keine allgemeingültige und formale Einigung über die Bedeutung aller Begriffe der in dieser Arbeit verwendeten objektorientierten Modellierung besteht (auch Atkinson vertritt in [Atk97] diesen Standpunkt), werden zu Beginn dieses Kapitels die wichtigsten Begriffe möglichst präzise eingeführt und mit der hier vorgestellten Bedeutung auch in dem Rest der Arbeit konsistent eingesetzt.

3.1 Objektorientierte Modellierung

Wie in Kapitel 2 bereits erläutert wurde, beschreibt ein Modell eine Abstraktion des modellierten Systems. Eine wichtige Form einer solchen Abstraktion ist die *Klassifikation* [Kas01][Boo91, S.137ff.], bei welcher *Typen*, auch *Konzepte* [OdR99] oder *Schemata* [WOI04] genannt, identifiziert werden.

Ein Typ weist dabei zwei wichtige Aspekte auf: *Intension* und *Extension* [StK02]. Bei der Intension („Inhalt“ [WOI04]) eines Typs handelt es sich um dessen Bedeutung oder vollständige Definition. In [GeW00] wird eine solche Definition z.B. durch ein Prädikat beschrieben, welches alle Elemente erfüllen, die von diesem Typ sind.

Die Extension („Umfang“ [WOI04]) eines Typs ist die Menge aller Elemente, für welche das Konzept zutrifft. So könnte man die Intension des Typs „Vierbeiner“ formulieren als „alles was vier Beine hat“. Eine mögliche Extension wäre dann

$$\mathbf{ext}(\text{Vierbeiner}) = \{\text{Susi, Strolch, Tom, Jerry}\}.$$

Ein Element e wird also als ein Typ T klassifiziert (oder e ist eine Ausprägung von T [WOI04]), wenn sowohl die Intension von T für e zutrifft als auch das Element e in der Extension von T beinhaltet ist [OdR99]. Letztere Bedingung kann man mit Hilfe obiger Mengenschreibweise auch als $e \in \mathbf{ext}(T)$ formulieren oder in der folgenden verkürzten Notation darstellen:

$$e@T \qquad \textbf{(Gleichung 3-1)}$$

Typischerweise betrachtet man bei der Modellierung von Software-Systemen die Repräsentation von Realweltobjekten durch *Objekte* in einem Software-System. Bei einem Realweltobjekt kann es sich dabei um ein physikalisches Objekt, wie z.B. einen Temperatursensor, oder um ein konzeptuelles Objekt, wie z.B. eine Benutzeranforderung, handeln. Oft gibt es aber keine scharfe Trennung zwischen diesen beiden Aspekten und viele Objekte sind sowohl physikalisch als auch konzeptuell einzuordnen [Tay99].

Neben den Objekten sind auch die Beziehungen zwischen diesen Objekten relevant und müssen demzufolge auch im Modell repräsentiert werden. In objektorientierten Modellen findet man daher (mindestens) zwei Arten von Typen: *Objekttypen* und *Beziehungstypen* (oder *Relationstypen*). Objekttypen klassifizieren dabei Objekte, wohingegen Relationstypen die Beziehungen zwischen diesen Objekten klassifizieren. Diese Typen und damit zusammenhängende objektorientierte Modellierungsbegriffe werden im weiteren Verlauf dieses Abschnitts vorgestellt.

3.1.1 Modellierung der statischen Struktur

Zunächst werden die wichtigsten Begriffe der objektorientierten Modellierung, die der Beschreibung der statischen Struktur dienen, vorgestellt.

Objekttypen und Klassen

Dass Objekttypen der Klassifikation von Objekten dienen wurde bereits herausgestellt. Typen sind dabei zunächst unabhängig von Implementierungs- oder Realisierungsaspekten (die obige Definition wurde ohne Referenz auf solche Aspekte beschrieben). Eine Möglichkeit zur objektorientierten Realisierung von Objekttypen bieten *Klassen*, welche eine generische Beschreibung der Instanzen des realisierten Typs im Sinne einer *Vorlage* oder eines „*Templates*“ darstellen (siehe [AKH00], [Tay99] und [GeW00]). Durch die *Instanziierung* einer Klasse erhält man dann Objekte des entsprechenden Typs.

Obwohl die Begriffe „Typ“ und „Klasse“ klar voneinander getrennt werden können ist dies bei dem Begriff „Objekt“ leider nicht der Fall, denn der Begriff kann sowohl die Instanz eines Typs als auch die Instanz einer Klasse beschreiben. Aus dem Modellzweck lässt sich aber die wahre Natur eines Objekts ableiten. So handelt es sich bei den Objekten in einem *Analysemodell* um Instanzen von Typen, bei einem *Entwurfsmodell* um Instanzen von Klassen [OdR99][Ode98, S.67].

Die Implementierung von Objekttypen durch Klassen lässt nun auch Spielraum bzgl. der konkurrenten Eigenschaften der Instanzen der Klasse. In der realen Welt sind alle Objekte üblicherweise *konkurrent*, wohingegen in einem Software-System einzelne Objekte oft sequentiell aktiviert werden. Objekte, die konkurrent zu allen anderen sind, realisiert man durch *aktive Klassen* (oder *Prozesse*), welche einen unabhängigen Kontrollfluss besitzen.

Assoziation, Aggregation und Komposition

Relationstypen, die eine (semantische) Beziehung zwischen einer Menge von Objekten repräsentieren, sind die *Assoziationen*. Dabei beschreibt eine Assoziation analog zu einem Objekttyp die Menge aller „Beziehungsinstanzen“ (engl. *Links*) mit gleicher Bedeutung. Neben einem Namen zur Identifikation der Assoziation und *Rollennamen* zur Kennzeichnung der *Assoziationsenden*, kann mit Hilfe der Angabe von *Multiplizitäten* die Anzahl der jeweils an einem Ende beteiligten Objekte eingeschränkt werden. Typische Multiplizitäten sind z.B. „0..1“, was maximal ein Objekt zulässt, oder „1..*“, was eine beliebige Anzahl von Objekten größer eins erlaubt [RJB99, S.346ff.].

Desweiteren kann man Assoziationen nach den zulässigen Möglichkeiten zum *Traversieren* (auch *Navigieren* oder *Verfolgen*) einordnen. Bei *gerichteten* Assoziationen ist dies nur in eine Richtung möglich, d.h. Objekte an der *Zielseite* der As-

soziation „sehen“ die Instanzen an *Ursprungsseite* der Assoziation nicht. *Bidirektionale* Assoziationen hingegen erlauben das Navigieren in beide Richtungen.

Die Anzahl unterschiedlicher Objekttypen, die an der Assoziation beteiligt sind ist prinzipiell nicht beschränkt. Oft verwendet man aber *binäre* Assoziationen (Anzahl beteiligter Typen ist zwei), die gegenüber den n-ären Assoziationen einfacher zu handhaben sind. Insbesondere ist die Beschreibung der Navigation bei n-ären Assoziationen komplex, da diese die Spezifikation von Mengen von Objekttypen, zwischen denen navigiert werden soll, erfordert [RJB99, S.350ff. und S.355].

Eine Spezialform der binären Assoziation ist die *Aggregation*, welche eine Beziehung zwischen dem „Ganzen“ und seiner Teile beschreibt. Dies impliziert, dass die Aggregations-Links keine Zyklen formen dürfen, weil sonst ein Objekt sowohl als „Ganzes“ als auch als Teil diesen „Ganzen“ interpretiert werden kann. Eine strengere Form der Aggregation ist die *strikte Aggregation*, bei welcher ein Teil immer nur von maximal *einem* „Ganzen“ aggregiert werden darf.

Die strengste Form der Aggregation ist die *Komposition*, die neben der strikten Eigenschaft, dass ein Objekt immer nur von *einem* „Ganzen“ aggregiert werden darf, zusätzlich fordert, dass die Lebenszeiten von Teil und „Ganzem“ zusammenfallen, d.h. anders als bei der Aggregation kann bei einer Komposition ein Teil nicht ohne sein „Ganzes“ existieren.

Es sei an dieser Stelle angemerkt, dass die Bedeutung von „Aggregation“ und „Komposition“ vielfach diskutiert wird; beispielhaft sei hier [StK02] und [Ode98] genannt. Henderson-Sellers und Barbier identifizieren in [HeB99] sogar 4608 theoretisch mögliche Varianten der Aggregationsrelation, von denen viele auch praktische Anwendbarkeit aufweisen. Aus diesen Gründen werden wir im Verlauf der Arbeit an der oben eingeführten etwas allgemeiner gehaltenen Definition und Interpretation dieses Relationstyps festhalten.

Attribute und Datentypen

Attribute (präziser wäre eigentlich der Begriff „Attributtypen“) beschreiben benannte und typisierte Platzhalter, die von den Instanzen der Objekttypen mit *Werten* belegt werden können. Man kann sie somit als eine spezielle Art der Darstellung der kompositionellen Beziehung des Objekts zu seinen (lokalen) Eigenschaften betrachten. Wichtig bei einer sauberen OO-Modellierung ist daher, Attribute stets als Beschreibung von Objektqualitäten (interne Eigenschaften) und Assoziationen als Spezifikation von Beziehungen zu anderen Objekten (externe Eigenschaften) zu verstehen [Ode98, S.4ff.][StK02].

Werte, die einem Attribut zugewiesen werden können, sind von einem bestimmten Typ. Typischerweise findet man als Typen für Attribute *primitive Typen* (oder auch „*Datentypen*“). Im Unterschied zu den Instanzen eines Objekttyps benötigen

die Instanzen eines primitiven Typs keine eindeutige Bezeichnung (z. B. durch einen Namen), da sie inhärent selbstidentifizierend sind (wie z. B. die ganzen Zahlen; vgl. auch [Atk97]). Zur Unterscheidung zwischen Datentypen und Objekttypen lassen sich charakteristische Eigenschaften wie Zustand, Identität oder Lebensdauer heranziehen [Poe00, S. 22f.].

Formal (mengentheoretisch) betrachtet, können sowohl Attribute als auch die oben vorgestellten binären Assoziationen (genauer deren Rollen) als Funktionen auf Typen interpretiert werden (siehe dazu auch [OdR99][KüS04]). Angenommen, für einen Objekttyp T_{O} wäre das Attribut a vom (Daten-)typ T_{D} definiert. Dann beschreibt a eine Abbildung

$$a : \mathbf{ext}(T_{\mathrm{O}}) \rightarrow \mathbf{ext}(T_{\mathrm{D}}) . \qquad \textbf{(Gleichung 3-2)}$$

Für jede konkrete Instanz des Objekttyps T_{O} wird (zu einem gegebenen Zeitpunkt) durch diese Abbildung dem Attribut a ein Wert (Instanz) aus $\mathbf{ext}(T_{\mathrm{D}})$ zugewiesen:

$$a(o) = d \text{ mit } o@T_{\mathrm{O}},\ d@T_{\mathrm{D}} .$$

Dasselbe gilt für eine (binäre) Assoziation zwischen zwei Objekttypen T_1 und T_2, deren Assoziationsenden bei T_i den Rollennamen r_i besitzen. Damit beschreibt r_1 bzw. r_2 die Abbildung

$$r_1 : \mathbf{ext}(T_1) \rightarrow \mathbf{ext}(T_2) \text{ bzw. } r_2 : \mathbf{ext}(T_2) \rightarrow \mathbf{ext}(T_1) . \qquad \textbf{(Gleichung 3-3)}$$

Anders als bei obigen Attributen können sich an einem Relationsende aber mehr als ein „Wert“ (Instanz) befinden (immer dann wenn eine Multiplizität größer eins angegeben wurde). Daher muss man die Definition aus Gl. 3-3 entsprechend erweitern. Eine Möglichkeit, die in [OdR99] vorgestellt wird, ist den Wertebereich der Funktion als Potenzmenge (Menge aller Teilmengen) zu definieren. Damit ergeben sich obige Gleichungen zu

$$r_1 : \mathbf{ext}(T_1) \rightarrow \mathbb{P}(\mathbf{ext}(T_2)) \text{ bzw. } r_2 : \mathbf{ext}(T_2) \rightarrow \mathbb{P}(\mathbf{ext}(T_1)) \qquad \textbf{(Gleichung 3-4)}$$

Generalisierung und Vererbung

Generalisierung unterscheidet eine objektorientierte Modellierung von der „einfacheren“ *objektbasierten* Modellierung. Die Generalisierung beschreibt eine taxonomische Beziehung zwischen einem generellerem Typ und einem spezielleren Typ [Bra83][Kas01] und dient somit einer genauen Einordnung und Beschreibung von Konzepten. Das Gegenteil der Generalisierung ist die *Spezialisierung* auch „*Subtyping*“ genannt [OdR99]. Dabei beschreibt der *Subtyp* eine Teilmenge der Extension des *Supertyps*, womit jede Instanz eines Subtyps auch gleichzeitig vom Supertyp ist.

Führen wir in unserem Beispiel vom Anfang den Subtyp „Hund“ ein, so bedeutet dies

$$\mathbf{ext}(\text{Hund}) \subseteq \mathbf{ext}(\text{Vierbeiner}) .$$

Die Extension von „Hund“ bestünde in unserem Beispiel dann aus der Menge $\{\text{Susi}, \text{Strolch}\}$. In Abb. 3-1 ist dieser Zusammenhang nochmals als Venn-Diagramm veranschaulicht.

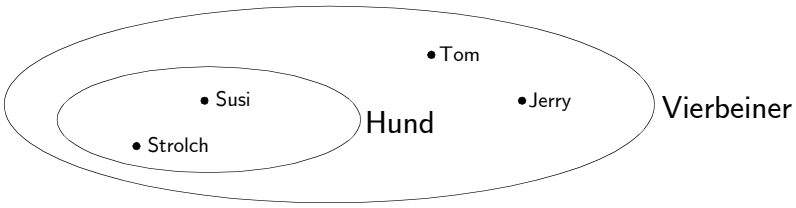


Abbildung 3-1. Spezialisierung von Typen (Darstellung der Extension)

Die Extensionen der Subtypen eines Supertyps sind dabei standardmäßig disjunkt [EJT04]. Im Beispiel hätte die Extension eines Typs „Katze“ keine Überschneidung mit der Extension von „Hund“.

Da die Generalisierung eine Beziehung zwischen Typen spezifiziert, ist sie auf der Instanzenebene nicht mehr *explizit* sichtbar (d.h. es existieren keine Generalisierungs-Links) und ist damit – anders als die Assoziation – kein Typ. Allerdings darf wegen der Beziehung zwischen Supertyp und Subtyp eine Instanz des spezielleren Elements auch da verwendet werden, wo eine Instanz des generelleren Elements erlaubt ist [Poe00, S.76].

Typischerweise findet man die Generalisierungsrelation zwischen Objekttypen, aber auch eine Generalisierung von Assoziationen (die ja wie bereits erwähnt auch Typen sind) ist durchaus sinnvoll. Wie bei der Spezialisierung von Objekttypen gilt auch für die Spezialisierung von Assoziationen, dass die speziellere Assoziation die Intension präzisieren (z.B. durch weitere Einschränkungen) und die Extension verkleinern muss. Bei Assoziationen bedeutet dies z.B. dass man die erlaubten Typen an den Assoziationsenden verfeinern oder die Multiplizitäten einengen kann (siehe dazu auch [RJB99, S.163f.]).

Der Begriff der „*Vererbung*“, der oft gleichbedeutend mit dem Begriff „Spezialisierung“ verwendet wird, beschreibt im Kontrast zu Letzterem lediglich die technische Realisierung der Spezialisierung [Tay99][OdR99][WKC01]. Dies erfolgt durch eine systematische Konstruktion der spezielleren Elemente durch eine Zusammensetzung der beschreibenden Fragmente in der Generalisierungshierarchie. Das heißt, dass die speziellere Klasse die Eigenschaften (Relationen, Attribute und Methoden) der *Oberklasse* erbt und diese auch überschreiben kann.

Bei diesem Überschreiben ist dann darauf zu achten, dass syntaktische Bedingungen eingehalten werden und auch, dass ein konformes Verhalten realisiert wird.

So müssen z.B. die Typen gleichbenannter Attribute zusammenpassen, d.h. der Typ des Attributs der spezielleren Klasse muss (minimal) eine Spezialisierung des Typs des Attributs der generelleren Klasse sein (es handelt sich hierbei um eine sog. *kovariante Beziehung* oder kurz *Kovarianz*). In der Programmiersprache Java wird sogar die Identität gefordert (siehe [Poe00, S.128ff.] für weitere Details).

Wird die Vererbung als Implementierung der Generalisierung eingesetzt und ein Typ besitzt mehr als einen generelleren Typ, dann kann eine dadurch implizierte *Mehrfachvererbung* zu Problemen führen. Zum Beispiel kommt es zu „Kollisionen“, wenn beide Oberklassen identische Attribute besitzen (weitere Probleme der Mehrfachvererbung werden in [Bla94] aufgeführt). Zur Vermeidung solcher Probleme und zur Vereinfachung der Generalisierungshierarchien werden wir daher in dieser Arbeit nur die einfache Vererbung (max. eine Oberklasse) verwenden.

3.1.2 Modellierung des Verhaltens und der dynamischen Struktur

Neben der statischen Struktur können in einem objektorientierten Modell auch das Verhalten der Objekte und dynamische Strukturänderungen beschrieben werden. Die wichtigsten Begriffe einer solchen Verhaltensmodellierung werden hier nun vorgestellt.

Operationen, Aktionen, Methoden und Interfaces

Operationen spezifizieren Transformationen oder Abfragen (allgemeiner Dienste), die von einem Objekt ausgeführt werden können. Operationen bestehen aus einzelnen *Aktionen*, welche implementierungsunabhängige Spezifikationen von atomaren Berechnungsanweisungen darstellen (siehe dazu [RJB99, S.122ff.] und [Gen99]). Die Ausführung einer Aktion und dementsprechend auch deren Ergebnis hängt von dem aktuellen *Zustand* des Objekts ab, welcher durch diese Ausführung auch geändert werden kann. Der Zustand eines Objekts lässt sich durch die Menge aller Attributbelegungen und Links beschreiben (andere gleichwertige Definitionen des Objektzustands sind in [Ode98, S.59ff.] erläutert). Damit ergibt die Menge der Zustände aller Objekte den Zustand des Gesamtsystems [Sch99, S.22].

Implementiert werden Operationen z.B. durch die *Methoden* einer Klasse. Die Definition einer Methode innerhalb einer Klasse bedeutet nun aber auch, dass die Klasse sowohl das „was“ als auch das „wie“ spezifiziert. Mit Hilfe von *Interfaces*, die nur die Signaturen der Operationen und nicht die Methoden selbst beschreiben, lassen sich diese beiden Aspekte sauber trennen. Mit einem Interface lässt sich also ein Typ beschreiben, ohne eine vollständige Klassendefinition vornehmen zu müssen. Dies hat insbesondere den Vorteil, dass auch Objekte von Klassen, die nicht in einer Vererbungsbeziehung zueinander stehen, gleichberechtigt verwendet werden dürfen, solange sie dasselbe Interface (und damit denselben Typ) besitzen [Tay99].

Ereignisse, Nachrichten und Signale

Beobachtbare und relevante Zustandsänderungen sind *Ereignisse* (engl. „*Events*“) [Ode98, S. 78]. Da die Ausführung einer Aktion – wie oben erläutert – eine Zustandsänderung bewirkt, verursacht eine solche Ausführung auch das Auftreten von Ereignissen. Umgekehrt kann ein Ereignis die Ausführung einer Operation (und dadurch die Ausführung von Aktionen) bedingen, wenn eine entsprechende *Auslöseregel* („*Trigger Rule*“ [MaO92, S. 94f.]) den Aufruf dieser Operation in Abhängigkeit vom auslösenden Ereignis spezifiziert. Demzufolge kann ein Ereignis von anderen Ereignissen abhängen oder selbst eine Kette anderer Ereignisse bewirken (siehe dazu auch [MaO92, S. 89ff.]).

Ein weit verbreiteter Formalismus zur Spezifikation dieser Verhaltensaspekte sind die *Endlichen Automaten* (oder „*Finite-State Machines*“). Diese ermöglichen eine explizite Beschreibung von Zuständen und Zustandsübergängen, welche *Transitionen* genannt werden. Die Auslösung dieser Übergänge kann durch die Angabe von Ereignissen spezifiziert werden. Aktionen können für eine Transition oder auch für einen Zustand angegeben werden. Im ersten Fall handelt es sich bei den damit beschriebenen Automaten um *Mealy-Automaten*, im letzten Fall um *Moore-Automaten*. Natürlich ist auch eine Kombination dieser beiden Ansätze möglich (siehe [Ode98, S. 85ff.]). Ein Endlicher Automat beschreibt somit jeweils die Reaktion *eines* Objekts auf externe „Stimuli“. In Abschnitt 6.2.3 auf Seite 151 werden wir nochmals auf den Formalismus der Zustandsautomaten zurückkommen.

So wie Objekte zu Objekttypen klassifiziert werden, kann man auch eine Klassifikation von Ereignissen zu *Ereignistypen* vornehmen. Beispiele für Ereignistypen sind die Änderung einer Attributbelegung oder das Einfügen eines neuen Links (weitere Ereignistypen sind in [MaO92, S. 88f.] aufgelistet). Besondere Ereignisse sind die Erzeugung oder Terminierung von Objekten. Dabei kann man die Erzeugung eines Objekts insofern als eine Zustandsänderung (die ein Ereignis impliziert) auffassen, als dass sich der Zustand von „kein Objekt sein“ zu „Objekt eines bestimmten Typs sein“ ändert. Bei einer Terminierung erfolgt die umgekehrte Zustandsänderung [Ode98, S. 79].

Bezüglich der Systemstruktur relevante Ereignisse (oder Aktionen, welche diese Ereignisse auslösen) sind das Hinzufügen oder Entfernen von Objekten und Links, da dadurch zur Laufzeit die Struktur beeinflusst werden kann und somit eine Modellierung der dynamischen Systemstruktur ermöglicht wird.

Zur Realisierung von Operationen ist oft eine *Interaktion* zwischen Objekten notwendig. So müsste man z. B. in einem Buchhaltungssystem zur Berechnung des Monatslohns eine Abfrage „liefere geleistete Stunden“ an alle Objekte der „Arbeiter“-Klasse senden. Eine solche Kommunikation erfolgt über den Austausch von *Nachrichten* [Tay99][RJB99, 333ff.], welche den Aufruf einer Operation im Empfänger

der Nachricht bewirken („*call action*“, [RJB99, S. 123]). Eine solche Nachricht kann das aufrufende Objekt entweder so lange blockieren bis das Ergebnis vorliegt (*synchroner Aufruf*) oder ohne Blockierung eine Weiterbearbeitung der aufrufenden Operation erlauben (*asynchroner Aufruf*) [DaC03, S. 109].

Neben einer Delegation von Diensten kann man solche Nachrichten auch nutzen, um die oben beschriebenen Ereignis-Ketten zu realisieren [CoD94, S. 153]. In diesem Fall wird die Nachricht durch eine sog. „*send action*“ [RJB99, S. 124] erzeugt. Diese Art der Nachricht wird auch häufig als *Signal* bezeichnet, so z. B. in der Modellierungssprache SDL (siehe [OFM94, S. 60ff.]).

3.1.3 Weitere Modellierungsbegriffe

Die oben vorgestellten Konstrukte zur Realisierung von Typen reichen oft nicht aus, um Objekte und deren Beziehungen vollständig zu beschreiben. So lässt sich zwar bei einer Assoziation der geforderte Typ an den Assoziationsenden spezifizieren, nicht aber zum Beispiel, dass ein und dieselbe Instanz eines Typs in zwei Links unterschiedlicher Assoziationen beteiligt sein muss. Zur Spezifikation solcher Einschränkungen bzgl. der Objekte führt man *Constraints* (auch *Invarianten* oder *Regeln*) ein. Eine zulässige Instanziierung eines Modells ist dann eine solche, in der alle Constraints erfüllt (d. h. „wahr“) sind. Ein Beispiel für ein Constraint wäre, dass kein „Hund“ älter als 50 Jahre werden kann:

$$\forall o@\text{Hund} : \text{alter}(o) \leq 50\,.$$

Ein weiteres Constraint haben wir mit dem Modellierungsbegriff der Multiplizität bereits kennengelernt. Die Multiplizität beschränkt die Anzahl der an einer Assoziation beteiligten Objekte und repräsentiert damit ein sog. *Kardinalitäts-Constraint* [Ode98, S. 33ff.]. Ein Beispiel für weitere solcher strukturellen Constraints findet sich in [AkK02], wo diese zur Präzisierung des Kerns einer OO-Modellierungssprache verwendet werden.

Neben solchen strukturellen Constraints (weitere Beispiele findet man in [CoD94, S. 46ff.]) lassen sich auch Constraints bzgl. des Verhaltens beschreiben. Üblicherweise erfolgt dies durch die Spezifikation von *Vor-* und *Nachbedingungen*. Eine Vorbedingung muss vor der Ausführung einer Operation gültig sein (dabei muss der Aufrufer der Operation die Gültigkeit der Bedingung prüfen [RJB99, S. 392]). Nach einer Operationsausführung ist die Nachbedingung erfüllt (wird die Nachbedingung nicht erfüllt, dann handelt es sich um eine fehlerhafte Implementierung der Operation [RJB99, S. 391]). Durch solche Constraints können daher auch Zeitanforderungen ausgedrückt werden, auf welche wir in Abschnitt 9.1.3 auf Seite 250 nochmals eingehen werden.

3.2 Metamodellierung

Betrachtet man die in einem Modell beschriebenen Konzepte, so lassen sich diese wiederum mit Hilfe der im letzten Abschnitt eingeführten Ansätze klassifizieren. Damit erhalten wir, wie bereits in Abschnitt 2.1.3 auf Seite 17 eingeführt, ein *Metamodell*. Solch eine „Meta"-Betrachtung ist übrigens mit jeder Begriffswelt möglich, die mächtig genug ist, Aussagen über sich selbst zu treffen (siehe dazu [Atk97]).

Interpretiert man nun ein Metamodell wiederum als ein zu modellierendes System (siehe Definition der Abstraktion in vorhergehendem Abschnitt), dann lässt sich dieses ebenfalls durch ein Metamodell beschreiben, was damit zu einem *Meta-Metamodell* wird. Zusammen mit dem Ausgangssystem und dessen Modell erhalten wir somit schon vier Ebenen der Modellierung. In Abb. 3-2 sind diese Ebenen (von denen es theoretisch unendlich viele geben kann) grafisch gezeigt.

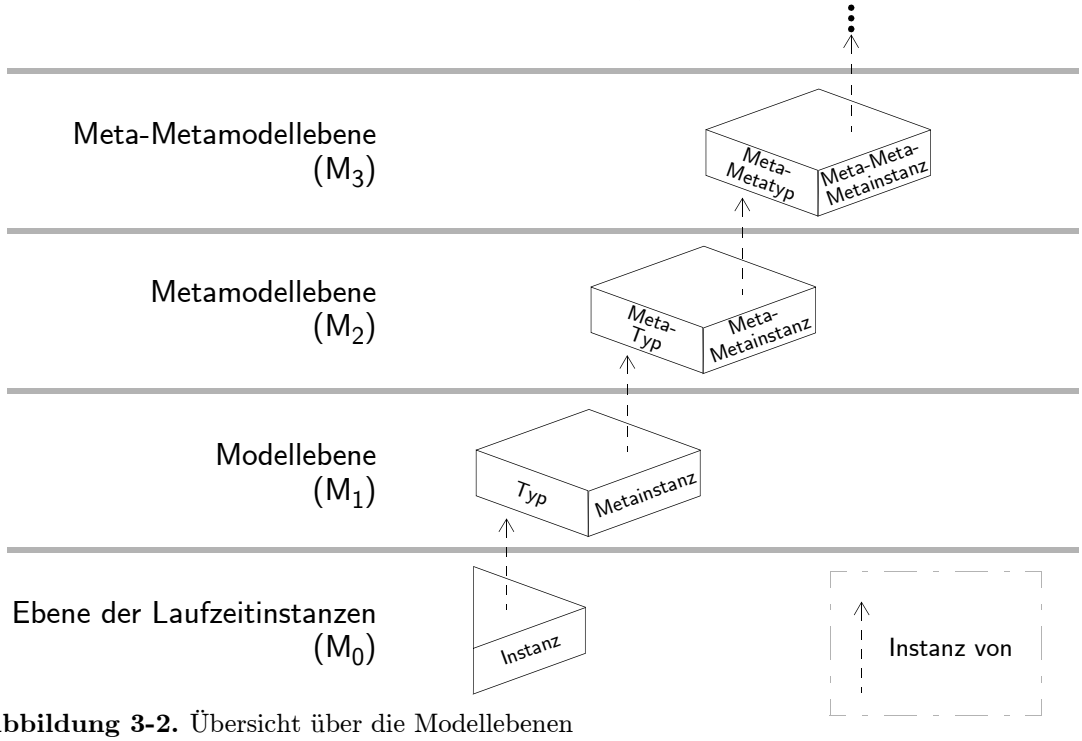


Abbildung 3-2. Übersicht über die Modellebenen

Dabei kann man allgemein jede Modellebene als eine Instanz der darüberliegenden Ebene betrachten. Für unseren Metamodellierungsansatz wollen dies weiter einschränken, indem wir wie in [Atk97] fordern, dass jedes Element eines Modells die Instanz exakt eines Elements des darüberliegenden Modells ist. Dies impliziert, dass Instanzen und deren Typen nie auf derselben Modellebene auftauchen dürfen. In unseren Augen erlaubt nur eine solche *strikte Metamodellierung* eine präzise und konsistente Einordnung und saubere Trennung der Metaebenen.

Der „Meta"-Begriff kann sowohl absolut („das Produktmodell ist ein Metamodell") als auch relativ („ein Meta-Metamodell ist ein Metamodell eines Meta-

modells“) interpretiert werden. Um eventuelle Verwechslungen zu vermeiden, führen wir daher analog zur „Four Layer Metadata Architecture“ der Object Management Group (OMG) [OMG02a] eine eindeutige Kennzeichnung der Ebenen ein. Die unterste Ebene bekommt dabei die Nummer 0, wobei mit jeder neuen Metaebene diese Zahl um eins erhöht wird. So entspricht dann z.B. die M_1-Ebene der Ebene der „normalen“ Software-Modelle (siehe dazu auch [Atk99]).

Neben dieser Nummerierung sind in Abb. 3-2 desweiteren die zwei Facetten der Modellierungsentitäten zu erkennen, die bei Vorhandensein von mehr als einer Modellebene auftreten können (in [AtK00] wird auf diese Thematik detailliert eingegangen). Auf der Ebene M_n handelt es sich bei solchen Entitäten zum einen um Typen, welche Instanzen der Ebene M_{n-1} beschreiben, zum anderen handelt es sich dabei um die Instanzen eines Metatyps von Ebene M_{n+1}. Solche Instanzen von Metatypen wollen wir im folgenden in Analogie zu der *Typ-Instanz Dichotomie* als *Metainstanzen* bezeichnen. Insbesondere bedeutet dies, dass der Begriff „Metaobjekt“ dann als Instanz eines Metaobjekttyps zu verstehen ist. Es sei darauf hingewiesen, dass abweichend von dieser Bedeutung der Begriff „Metaobjekt“ von der OMG als Oberbegriff aller Metaentitäten (Metarelation, Metaattribut, etc.) benutzt wird (siehe dazu [RJB99, S. 340]).

Nicht alle Elemente einer Modellebene müssen allerdings diese Dualität besitzen. Es gibt also Metainstanzen, die keinen Typ darstellen. So kann man in einem M_1-Modell sicherlich Metaobjekte für einzelne Benutzeranforderungen finden, die in dem endgültigen Software-System nie als Objekte auftauchen.

Es ist sehr wichtig an dieser Stelle nochmals auf den Unterschied zwischen Instanziierung und Generalisierung hinzuweisen. Bei mehr als einer Modellebene ist es nämlich erstmals möglich, dass eine Instanz durch die mehrfache Instanziierung eines Metatyps entsteht. Eine Instanz auf M_1 kann z.B. die Instanz eines M_2-Typs sein, der selbst wieder die Instanz eines M_3-Typs ist. Bei einer oberflächlichen Betrachtung des Instanziierungsbegriffs – typischerweise durch eine sprachliche Formulierung wie „eine Instanz *ist-ein* Typ“ [Bra83] – scheint damit auch die Instanz auf M_1 eine Instanz des M_3-Typs zu sein. Dies ist allerdings nicht der Fall, da die Instanziierungsrelation (anders als die Generalisierung) *nicht* transitiv ist (siehe dazu [AKH00] und [AKH03]). Ein anschauliches Beispiel dafür bieten die folgenden Aussagen [Küh03a]:

1. „Peter“ *ist-ein* „Bäcker“
2. „Bäcker“ *ist-ein* „Beruf“

Wäre die Instanziierung, die durch *ist-ein* beschrieben wird, transitiv, dann würde man folgern:

3. „Peter“ *ist-ein* „Beruf“

Dies ist offensichtlich falsch, da „Peter“ kein „Beruf“ sein kann sondern eine „Person“ ist.

Ein weiteres Argument gegen diese Transitivität der Instanziierung ist, dass die Attribute eines Objekttyps bei dessen Instanziierung mit Werten belegt werden und deshalb nicht mehr zur Verfügung stehen (dasselbe gilt für die Assoziationen). Würde man die Instanziierung also nutzen um Folgendes zu beschreiben: „Peter“ *ist-ein* „Mann“, „Mann“ *ist-ein* „Mensch“ und „Mensch“ hätte das Attribut „alter“, dann würde „Mann“ nicht das Attribut „alter“ sondern lediglich eine Attributinstanz (Wertbelegung) besitzen, die für alle „Männer“ gleich ist. Demzufolge könnte „Peter“ nie über ein eigenes Alter verfügen.

Formal lässt sich diese Eigenschaft mit Hilfe der mengentheoretischen Definition der entsprechenden Begriffe beweisen. Nehmen wir an, dass e von einem bestimmten Typ T sei, also $e@T$ gilt. Handelt es sich bei ST um dessen Supertyp, dann gilt $\mathbf{ext}(T) \subseteq \mathbf{ext}(ST)$ und demzufolge auch $e@ST$. Sei MT allerdings ein Metatyp von T, dann gilt $T@MT$ und damit wäre eine Extension von MT, z.B.

$$\mathbf{ext}(MT) = \{\{e\}\}$$

und folglich gilt

$$e \notin \mathbf{ext}(MT)\,.$$

□

3.2.1 Modellebenen M_0 bis M_3

Um ein Gefühl für die verschiedenen Modellebenen und die möglichen Alternativen bei einer Metamodellierung zu bekommen werden in den folgenden Unterabschnitten die in Abb. 3-2 auf Seite 38 eingeführten Ebenen im Detail behandelt und jeweils Beispielmodelle für jede dieser Ebenen präsentiert. Zunächst werden wir zur Vorstellung dieser Ebenen die „traditionelle“ Form der Metamodellierung (so wie sie z.B. auch zur Definition der UML [OMG03] eingesetzt wird) verwenden. In Kapitel 4 werden wir dann auf mögliche Alternativen eingehen.

Ebene der Laufzeitinstanzen (M_0)

Die unterste Ebene in der Modellhierarchie beinhaltet die *Laufzeitinstanzen*, welche bei der Ausführung der Software erzeugt, verändert oder vernichtet werden. Handelt es sich bei dem M_1-Modell nicht um ein Software-Modell sondern um ein Datenmodell (oder ein Datenbankschema), dann sind hier die eigentlichen „Daten“, die in der Datenbank abgelegt sind, zu finden. Demzufolge wird diese Ebene auch als *Informationsebene* bezeichnet (siehe z.B. [OMG02a]).

An dieser Stelle erscheint es sinnvoll, den Unterschied zwischen diesen Instanzen und einem Instanzenmodell (wie z.B. dem UML Objektdiagramm) deutlich zu ma-

chen. Bei den in solchen Modellen spezifizierten Objekten handelt es sich keinesfalls um Elemente der Ebene M_0. Vielmehr beschreibt eine Menge solcher Modellobjekte einen Schnappschuss oder eine Momentaufnahme der eigentlichen Laufzeitinstanzen. Solch ein Objektmodell ist also selbst wieder als Abstraktion der Ebene M_0 einzuordnen (und würde zwischen den bisherigen Ebenen M_0 und M_1 liegen).

Als anschauliches Beispiel für die im Folgenden dargestellten Modellebenen wollen wir ein hypothetisches 2D-Grafikprogramm betrachten, das komplexe grafische Elemente (bestehend aus Polygonen und Kreisen) zeichnen kann. In Abb. 3-3 ist ein möglicher Bildschirmabzug eines solchen Werkzeugs gezeigt. Wie man erkennt, sind sowohl die Häuser als auch die Bäume aus weniger komplexen grafischen Elementen (Rechteck, Dreieck, Kreis) zusammengesetzt.

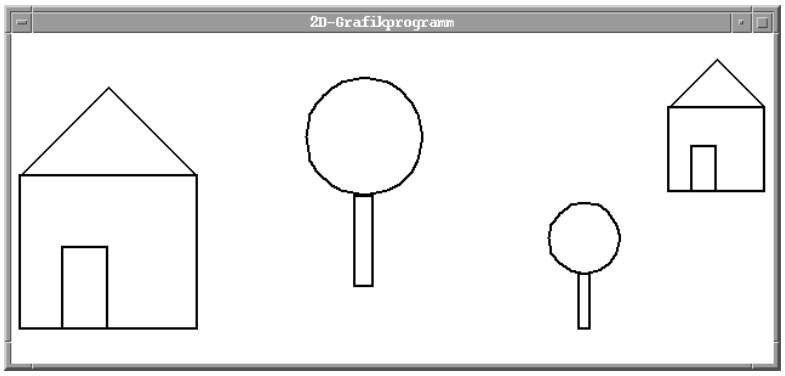


Abbildung 3-3. M_0-Ebene – Bildschirmabzug zeigt Instanzen grafischer Elemente

Modellebene (M_1)

Die Ebene M_1, die „eigentliche" Modellebene, stellt die erste Ebene der Abstraktion von den Laufzeitinstanzen (also die Klassifikation dieser Instanzen) dar. Bei Software-Systemen handelt es sich bei Modellen dieser Ebene um Software-Modelle oder -Spezifikationen. Bei Datenbanksystemen ist hier die Bezeichnung der *Metadaten*-Ebene zu finden, da man eine Klassifikation der Daten (oder auch „Daten über Daten") beschreibt.

Nehmen wir unser obiges Beispiel des grafischen Werkzeugs, so können wir folgende Typen von grafischen Elementen identifizieren (siehe Abb. 3-4): Dreieck, Rechteck, und Kreis und die daraus zusammengesetzten komplexeren Typen Haus und Baum. Wie man an dem Objekttypdiagramm in Abb. 3-4 leicht erkennt, besteht damit jedes Haus aus exakt einem Dreieck (dem Dach) und einer Etage, die aus zwei Rechtecken besteht.

Jeder Baum besteht aus einem Rechteck (dem Stamm) und einem Kreis (dem Wipfel). Dies wird jeweils durch eine Aggregationsrelation zwischen den beteiligten Typen beschrieben (grafisch ist dabei das „Ganze" durch einen offenen Diamanten

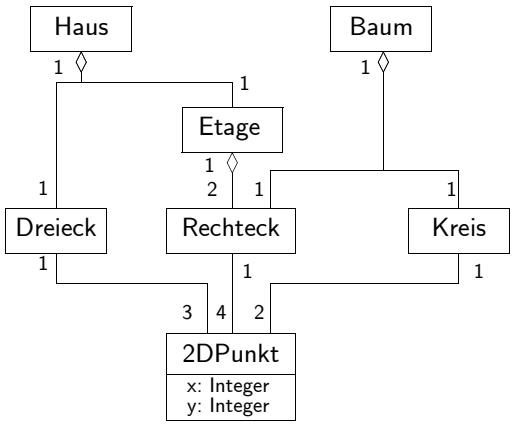


Abbildung 3-4. M_1-Ebene – Modell komplexer grafischer Elemente

gekennzeichnet). Zur grafischen Darstellung eines OO-Modells nutzen wir in dieser Arbeit die Notation der UML Klassendiagramme (siehe z.B. [RJB99, S.41ff.]).

Jedes Dreieck lässt sich durch seine drei Ecken spezifizieren, die in unserem zweidimensionalem Programm jeweils durch eine x- und y-Koordinate auf dem Bildschirm gegeben sind. Dies wird durch die ganzzahligen (Integer) Attribute x und y des Objekttyps 2DPunkt beschrieben. Für jedes Rechteck benötigen wir demzufolge vier solcher Ecken, für Kreise genügen zwei (ein Mittelpunkt und ein Punkt auf dem Kreisumfang). Die Assoziation von den Typen der grafischen Grundelemente zu dem Objekttyp 2DPunkt trägt diesem Sachverhalt Rechnung.

Eine mögliche Abstraktion, die eine Einordnung der im M_1-Modell vorhandenen Elemente erlaubt, ist die Generalisierung. In Abb. 3-5 sind mögliche Supertypen dargestellt. Es wurden dabei zwei Haupttypen von Grafikelementen identifiziert, SimplesGrafikElement und KomplexesGrafikElement.

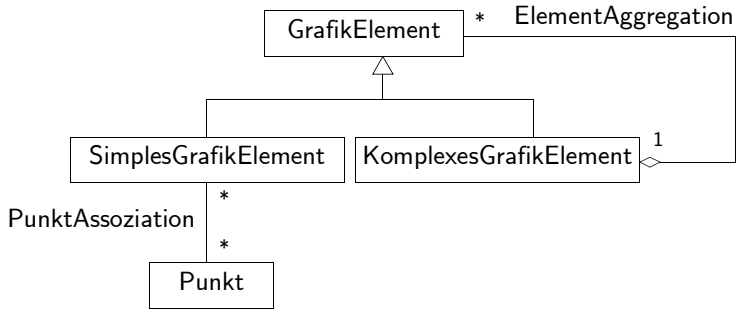


Abbildung 3-5. M_1-Ebene – Generalisierung von Grafikelementen

Sowohl bei SimplenGrafikElementen als auch bei KomplexenGrafikElementen handelt es sich um GrafikElemente, was durch die Generalisierungsrelation (grafisch durch eine offene Pfeilspitze gekennzeichnet) beschrieben wird.

Ein komplexes Grafikelement kann nun entweder aus simplen Elementen oder aber aus weniger komplexen Elementen bestehen. Diesen Sachverhalt modelliert

man durch die Aggregationsrelation (ElementAggregation) von KomplexesGrafikElement zu GrafikElement. Analog erfolgt die Modellierung der Beziehung zwischen einem simplen Grafikelement und den seine Form beschreibenden Punkten durch eine Assoziationsrelation (PunktAssoziation) von SimplesGrafikElement zu Punkt.

Nach der Definition dieses generelleren M_1-Modells können wir jetzt das Beispiel aus Abb. 3-4 auf Seite 42 auch als Spezialisierung dieses Modells darstellen. In Abb. 3-6 ist dies grafisch gezeigt.

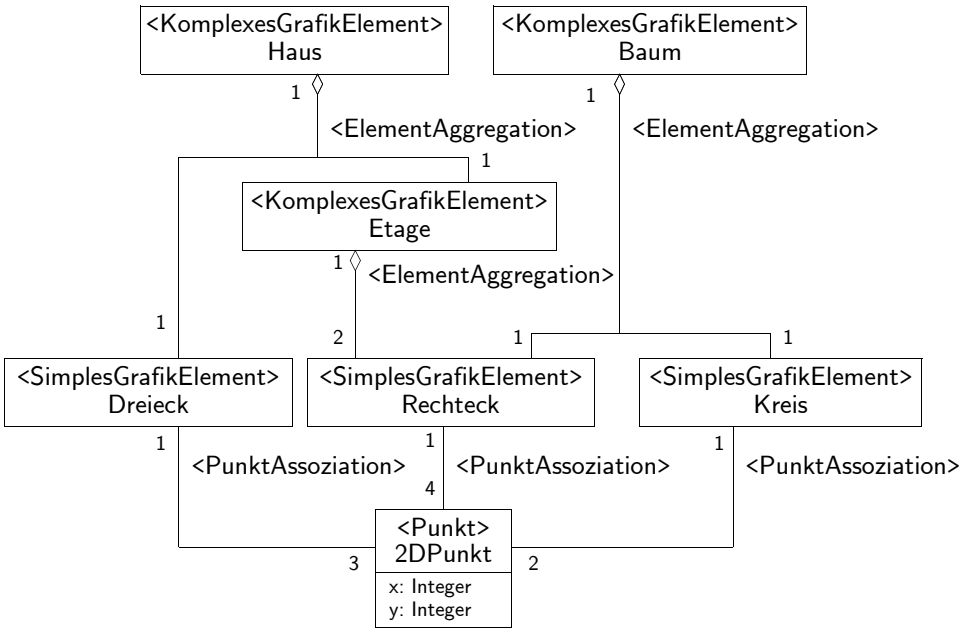


Abbildung 3-6. M_1-Ebene – Modellierung von Grafikelementen mittels Generalisierung

Dabei stellt die Notation <Supertyp> die Generalisierungsrelation (sowohl für Objekttypen als auch für Assoziationen) in einer kompakten Form dar. Wir orientieren uns hierbei an einem Vorschlag von Atkinson et al. [AKH03], da dieser gegenüber der Verwendung der Stereotyp-Notation (welche „korrekt“ zur Beschreibung eines virtuellen Metatyps auf Modellebene dient [RJB99, S. 449]), die klarere Modellierung erlaubt.

Dieser typische Einsatz der Generalisierung (siehe [RJB99, S.163]), den man häufig auch in Grafikbibliotheken findet, bringt allerdings auch Probleme mit sich:

1. Die Modellierer müssen sich an Konventionen halten, die nicht im Modell beschrieben sind. Dazu gehört z. B. dass zur Spezifikation eines spezielleren Typs eines komplexen Grafikelements keine neue Aggregationsrelation eingeführt werden darf, sondern die ElementAggregation verfeinert werden muss.
2. Man muss vorschreiben (z. B. durch ein Constraint), dass für ein Grafikelement alle PunktAssoziationen nur zu einer Art von Punkt führen, weil es sonst zu Problemen der grafischen Darstellung kommt (z. B. wenn eine Ecke eines zweidimensionalen Dreiecks im 3D-Raum liegen würde).

3. Ein solches Modell ist bzgl. seiner (dynamischen) Erweiterbarkeit ungenügend (siehe [Ode98, S. 23ff.][KüS04]). Mit jedem Einführen eines neuen Typs von Grafikelement (z. B. eines Pentagons) muss dieses als Typ der Ebene M_1 (mit zugehöriger Generalisierungs- und Aggregationsrelation) eingeführt werden. Eine dynamische Erweiterung ist damit nicht möglich.

Eine alternative Art der Modellierung, die eine Beseitigung der obigen Probleme ermöglicht, stellt die Einführung einer expliziten Metamodellebene dar. Diese wird im folgenden Abschnitt erläutert.

Metamodellebene (M_2)

Mit Hilfe einer weiteren Modellebene, der Metamodellebene (M_2), können wir die Elemente eines M_1-Modells klassifizieren. Wie für die Ebene M_1 können wir auch für diese Ebene wieder dieselben objektorientierten Ansätze anwenden. Betrachtet man die M_1-Modelle als Entwicklungsprodukte in der Software-Entwicklung, dann handelt es sich bei dem M_2-Modell um ein Produktmodell, das diese Produkte beschreibt (siehe Abschnitt 2.2.1 auf Seite 19).

Wie im vorigen Abschnitt schon angekündigt wurde, erlaubt uns diese Ebene auch eine alternative Modellierung unseres Grafikbeispiels. Klassifiziert man die Elemente aus Abb. 3-4 auf Seite 42, so lassen sich zwei verschiedene Typen von Grafikelementen identifizieren: solche, deren Instanzen simple Grafikelemente sind (Dreieck, etc.), und solche, deren Instanzen komplexe Elemente darstellen (z. B. Haus). Demzufolge führen wir in unserem Metametamodell die zwei Objekttypen SimplerGrafikElementTyp und KomplexerGrafikElementTyp ein, welche beide von dem allgemeineren Supertyp GrafikElementTyp abgeleitet sind (siehe Abb. 3-7).

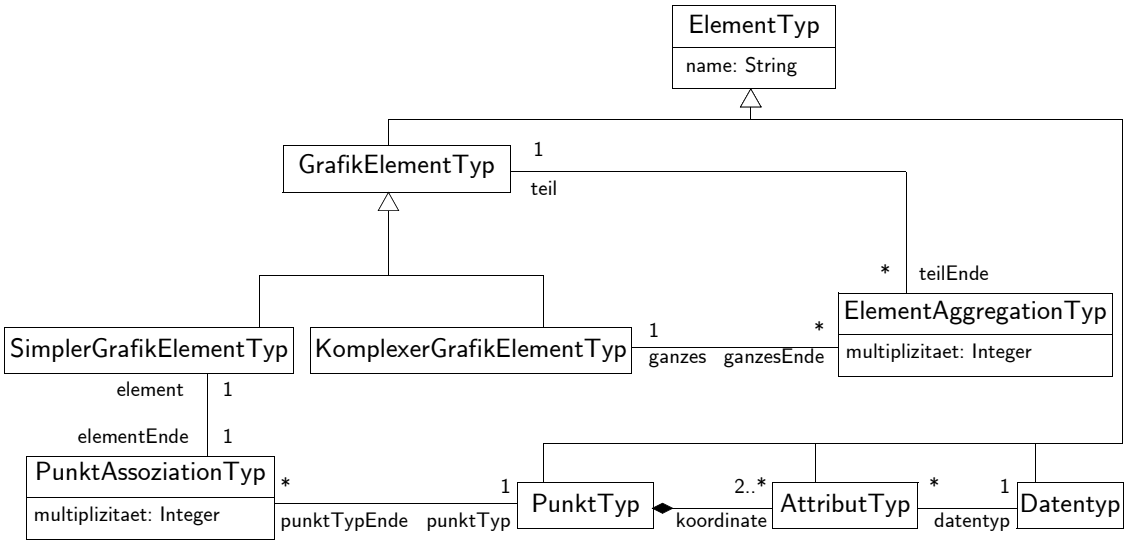


Abbildung 3-7. M_2-Ebene – Klassifikation der Elemente von Ebene M_1

Neue GrafikElemente auf M_1 lassen sich nun durch eine *Instanziierung* des entsprechenden GrafikElementTyps erzeugen. Dies stellt eine Lösung für das letzte Problem des vorhergehenden Abschnitts dar, da wir mit Hilfe der Instanziierung jetzt dynamisch z.B. den Grafikelementtyp Pentagon einführen können, indem wir den Typ SimplerGrafikElementTyp instanziieren.

Desweiteren lassen sich auf dieser Ebene auch die auf M_1 zulässigen Relationen zwischen den Modellelementen beschreiben, was die ersten beiden Probleme behebt. Den Sachverhalt, dass auf M_1 eine Aggregationsrelation zwischen einem komplexen Grafikelement und einem allgemeinen Grafikelement besteht, modelliert man durch eine Assoziation von KomplexerGrafikElementTyp zu dem Objekttyp ElementAggregationTyp, dessen Instanzen folglicherweise ElementAggregationen sind. Von diesem Typ aus führt wiederum eine Assoziation zum Supertyp GrafikElementTyp. Da sowohl auf der Seite von KomplexerGrafikElementTyp als auch auf der Seite von GrafikElementTyp eine Multiplizität von eins gefordert ist, realisiert dies eine binäre Assoziation auf Ebene M_1. Mit dem multiplizitaet-Attribut beschreibt man die zulässige Anzahl der aggregierten M_0-Objekte (als Datentyp genügt dabei Integer, da für die Grafikelemente keine „offenen“ Multiplizitäten, also „*“, erlaubt sind). Ein entsprechendes Attribut für die „Ganzes“-Seite wird für dieses Modell nicht benötigt, da wir implizit von einer Multiplizität von eins ausgehen und somit nur 1-zu-n Aggregationen zulassen. (Eine allgemeinere Modellierung von Assoziationen wird bei der M_3-Ebene auftauchen.)

Analog erfolgt die Modellierung der Beziehung zwischen simplen Grafikelementen und den die Form beschreibenden Punkten. Hier wird die Klasse PunktAssoziationTyp definiert und zusätzlich die Multiplizität für das Ende der Assoziation von SimplerGrafikElementTyp zu PunktAssoziationTyp auf der Seite von PunktAssoziation auf eins gesetzt, was auf der M_1-Ebene die Verbindung eines Grafikelements zu lediglich einem Typ von Punkt erlaubt.

Letztlich muss noch der einzelne PunktTyp genauer beschrieben werden, was durch eine Komposition einer beliebigen Anzahl von AttributTypen abgebildet wird. So wird es für unseren 2DPunkt (auf Ebene M_1) zwei Attribute geben, die jeweils die x- und y-Koordinate des Punktes beschreiben. Zusätzlich wird der Datentyp dieser Koordinaten über eine Assoziation von AttributTyp zu Datentyp beschrieben.

Zur Vereinfachung dieses und aller folgenden Modelle wollen wir annehmen, dass die Standarddatentypen (Integer, Real, String, Boolean, etc.) auf jeder Modellebene zur Beschreibung der Typen von Attributen zur Verfügung stehen, was die explizite Definition dieser Typen für jede Ebene erspart und dadurch die Modelle vereinfacht.

Zuletzt definieren wir zusätzlich den Supertyp ElementTyp, damit allen Metainstanzen (und damit den Modellelementen auf M_1) ein Name (z.B. „Pentagon“) zugewiesen werden kann.

Mit den obigen Metatypen als Ausgangspunkt kann nun das M_1-Beispielmodell mit Hilfe von Metainstanzen dargestellt werden (siehe Abb. 3-8). In der UML findet man für die Kennzeichnung von Metainstanzen häufig das Stereotyp-Modellelement. Da dies aber unterschiedliche Anwendung findet (siehe [AKH03]), verwenden wir hier in Anlehnung an Gl. 3-1 auf Seite 30 die „@“-Notation, um Verwechslungen zu vermeiden.

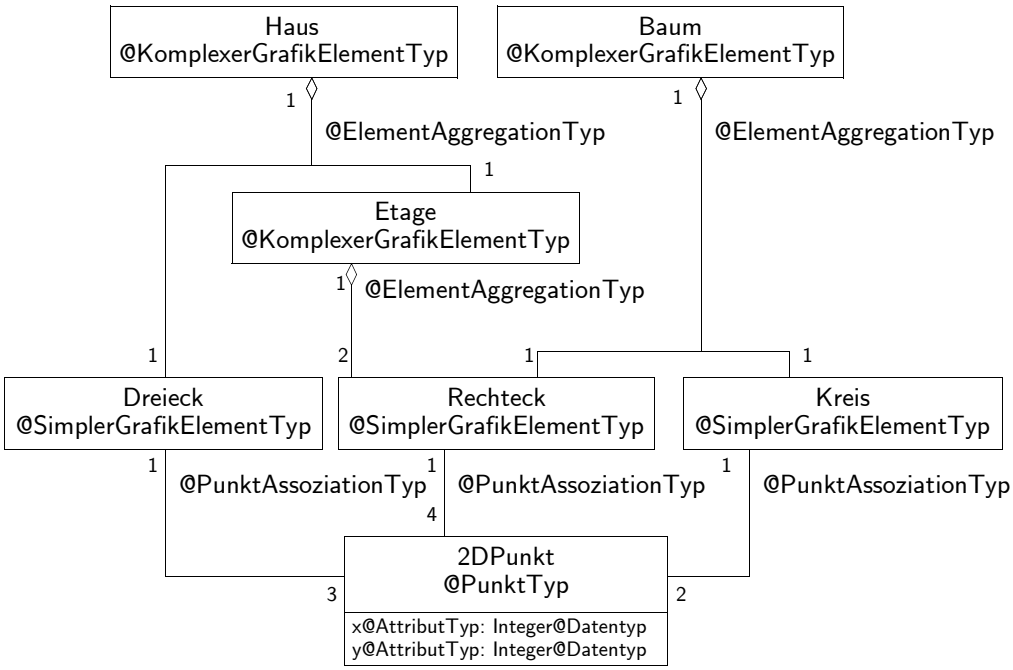


Abbildung 3-8. M_1-Ebene – Modellierung von Grafikelementen mittels Metainstanzen

Betrachtet man die bisher vorgestellten zwei Möglichkeiten zur Modellierung, dann stellt sich die Frage, welchen Unterschied es z.B. zwischen einem Subtyp Pentagon von GrafikElement auf M_1 und einer Instanz Pentagon des Metatyps GrafikElementTyp gibt. Die Wahrheit ist, dass es sich hierbei um ein und dieselbe Entität handelt. Denn wie der Name GrafikElementTyp schon suggeriert handelt es sich bei dessen Instanzen um GrafikElemente. Ebenso sind die Subtypen von GrafikElement per Definition GrafikElemente. Nicht von ungefähr sind sich die Modelle in den Abbildungen 3-6 und 3-8 so ähnlich. (Ein analoges Beispiel mit Bäumen und Baumarten findet sich in [Ode98, S.27f.].)

Die oben beschriebene Modellierung von Assoziationen und Attributen der Ebene M_1 scheint „indirekt“ zu sein. Sie ist aber aus zwei Gründen notwendig. Zum einen muss man im Falle der Assoziation zusätzliche Attribute (wie z.B. Multiplizitäten oder Rollennamen) beschreiben können. Zum anderen erscheinen bei der „traditionellen“ Metamodellierung alle Assoziationen bzw. Attribute, die in M_{n+1} spezifiziert werden, bei der Instanziierung in M_n als Links bzw. Attributinstanzen (mit

entsprechenden Wertebelegungen) und können daher nicht weiter instanziiert werden.

Noch deutlicher wird das Problem einer solchen „traditionellen“ Metamodellierung, wenn wir uns für die Modellierung mit Hilfe einer Metaebene entschieden haben und nun spezifizieren möchten, dass jedes Grafikelement eine eigene Farbe besitzen kann. Dies impliziert konsequenterweise die Definition eines Attributs farbe für jeden Typ von Grafikelement auf Ebene M_1. Auf der Ebene M_2 lässt sich dieses allerdings nicht beschreiben. Definiert man nämlich in GrafikElementTyp ein Attribut farbe, dann wird dieses bei der Instanziierung mit einem Wert belegt, d.h. eine Instanz von GrafikElementTyp, z.B. Pentagon, besitzt eine feste Wertzuweisung. Einzig mögliche Alternative wäre zunächst also die zusätzliche Einführung eines Supertyps auf M_1, der dieses Attribut definiert, und dann alle Grafikelemente davon abzuleiten. Damit erhalten wir aber wieder das Problem, dass die Konvention eingehalten werden muss, alle Instanzen von GrafikElementTyp auch von GrafikElement abzuleiten.

Die obigen Probleme werden wir in Abschnitt 3.2.3 nochmals detailliert diskutieren und Kapitel 4 eine Verbesserung der Multiebenenmodellierung vorschlagen.

Meta-Metamodellebene (M_3)

Betrachtet man sich das Metamodell aus dem vorangegangenen Abschnitt, dann könnte man auch sagen, dass wir damit eine „Sprache“ zur Beschreibung von Modellen von Grafikelementen spezifiziert haben. Demzufolge kann man nun auch ein Metamodell einsetzen, um die allgemeine „Sprache“ von objektorientierten Modellen zu beschreiben.

In unserem Grafikbeispiel ist dieses Metamodell sinnvollerweise auf der Ebene M_3 anzusiedeln, denn eine weitere Klassifikation der in Abb. 3-7 auf Seite 44 beschriebenen Elemente führt zweifelsohne zu den in Abschnitt 3.1 auf Seite 30 eingeführten objektorientierten Begriffe wie Objekttyp, Attribut, Generalisierung und Assoziation.

Da wir in unserem Beispiel nicht von allen OO-Begriffen Gebrauch machen (z.B. beschreiben wir kein Verhalten), stellt sich das M_3-Modell für unser Beispiel relativ einfach dar (siehe Abb. 3-9). Im Vergleich dazu beinhaltet z.B. der Kern des Metamodells von UML bereits 40 Objekttypen (UML 1.4/Core, siehe [OMG01, S.5-1ff.]).

Den Mittelpunkt unseres Metamodells stellt der GeneralisierbareElementTyp dar. Die Generalisierung wird durch eine reflexive Assoziation beschrieben, deren beide Assoziationsenden an dem GeneralisierbarenElementTyp enden. Der Supertyp und der Subtyp werden durch die entsprechenden Rollennamen ausgezeichnet. Wie schon in Abschnitt 3.1.1 auf Seite 31 eingeführt, wollen wir nur die einfache Spezi-

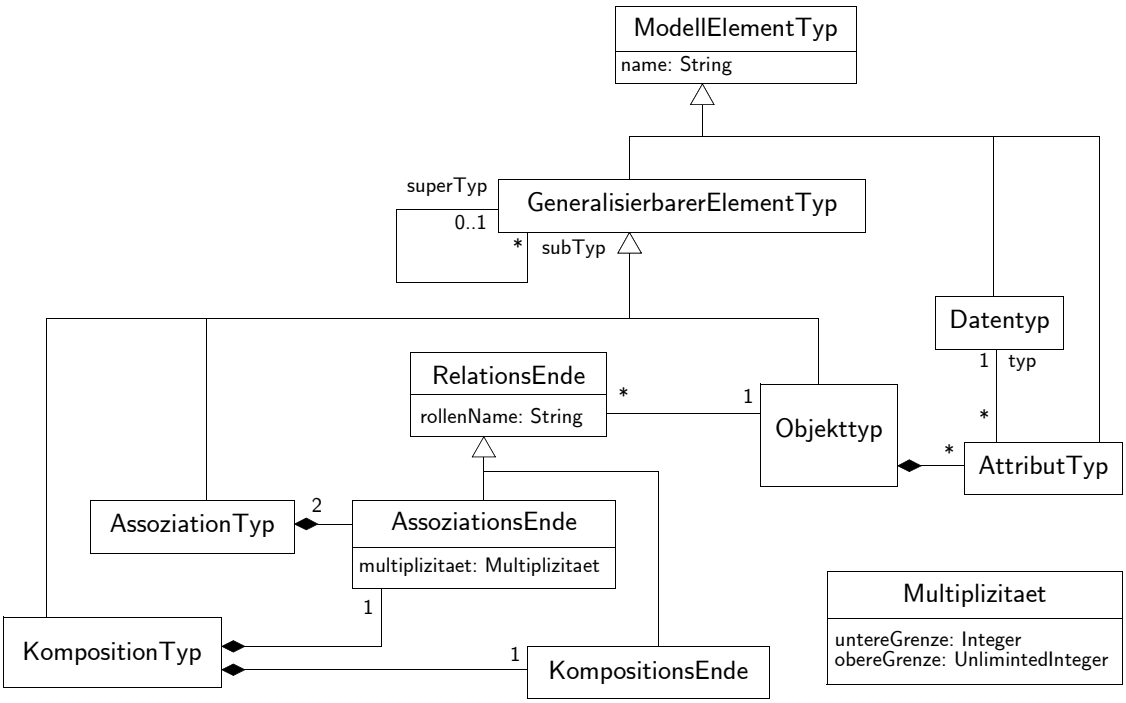


Abbildung 3-9. M_3-Ebene – mögliches Metamodell für objektorientierte Modelle

alisierung (max. ein Supertyp) in unseren Modellen verwenden. Dies wird durch eine Multiplizität von „0..1“ auf der superTyp-Seite der Assoziation modelliert.

Zwei Arten von generalisierbaren Elementen existieren: Objekttypen und AssoziationTypen (bzw. KompositionTypen). In unserem Beispiel umfasst Objekttyp alle durch UML-Klassensymbole dargestellten Modellentitäten auf Ebene M_2.

Zwischen den Instanzen dieses Objekttyps sind die Assoziation und die Komposition erlaubt (z.B. die Komposition zwischen PunktTyp und AttributTyp auf M_2). Dies wird in Abb. 3-9 durch den Objekttyp AssoziationTyp beschrieben, der aus den jeweiligen AssoziationsEnden besteht. Wir führen hier anders als auf Ebene M_2 explizite Elemente für die Relationsenden ein, um diesen einen Rollennamen (Attribut rollenName) zuweisen zu können.

Durch eine Festlegung auf genau zwei AssoziationsEnden pro AssoziationTyp (Multiplizität der Komposition zwischen Assoziation und AssoziationsEnde ist „2“) modellieren wir binäre Assoziationstypen. Die Komposition interpretieren wir als spezielle Form der Assoziation (siehe dazu auch Abschnitt 3.1.1 auf Seite 31), indem wir ein Ende der Relation durch einen besonderen Endpunkt KompositionsEnde auszeichnen. Anders als ein Assoziationsende besitzt dieser Objekttyp kein multiplizitaet-Attribut, da auf der „Ganzes“-Seite der Aggregation immer eine „1“ gefordert ist.

Der Typ des multiplizitaet-Attributs des AssoziationsEndes ist ein Objekttyp Multiplizitaet, der eine untere und eine obereGrenze beschreibt. Wir modellieren die Multiplizität als Attribut des Assoziationsendes, da es sich hierbei um eine Qualität des

Endes und nicht um eine Beziehung zu anderen Objekten handelt (siehe Diskussion der Attribute in Abschnitt 3.1.1 auf Seite 31). Zur Beschreibung der oberen Grenze führen wir analog zum UML Metamodell (siehe z.B. [OMG01, S.5-10]) einen neuen Datentyp UnlimitedInteger ein, der ein zusätzliches Literal für die unbegrenzte Kardinalität besitzt, also UnlimitedInteger = Integer $\cup$ {*}.

Die möglichen Attribute eines M_2-Elements beschreiben wir durch eine Kompositionsrelation von Objekttyp zu AttributTyp. Als Typ eines Attributs sind dabei nur Datentypen (primitiven Typen) vorgesehen, was durch die Assoziation zwischen AttributTyp und Datentyp beschrieben wird.

Zuletzt führen wir einen Supertyp ModellElementTyp ein, der eine Identifikation (Benennung) der Instanzen auf M_2 durch eine Wertbelegung des Attributs name erlaubt.

3.2.2 Weitere Ebenen

Mit der oben beschriebenen Ebene kann man nun in der Modellierung nicht einfach aufhören, denn auch die Konzepte dieser Ebene müssen irgendwo beschrieben werden. Am „einfachsten“ wäre dies durch das Einführen einer weiteren Metaebene, in dem Fall des Grafikbeispiels einer M_4-Ebene, möglich. Aber auch diese Ebene muss man wieder beschreiben. Damit sind theoretisch unendlich viele Metaebenen nötig.

Reflexive Beschreibung

Es bieten sich zwei Lösungen dieses Dilemmas an. Die erste Lösung basiert auf der Tatsache, dass sich ab einer gewissen Ebene die Konzepte nicht mehr voneinander unterscheiden. So würde man in einer hypothetischen Ebene M_4 wiederum die Konzepte Objekttyp, Relationstyp und Attributtyp definieren, da diese auf Ebene M_3 instanziiert werden. Dasselbe gilt für eine weitere Ebene M_5.

Konsequenterweise kann man die Ebene M_3 dann auch mit sich selbst beschreiben. Dieser *reflexive Ansatz*, wie er z.B. auch von der OMG zur Definition von MOF (siehe [OMG02a, S.2-4]) eingesetzt wird, ist in Abb. 3-10 grafisch veranschaulicht.

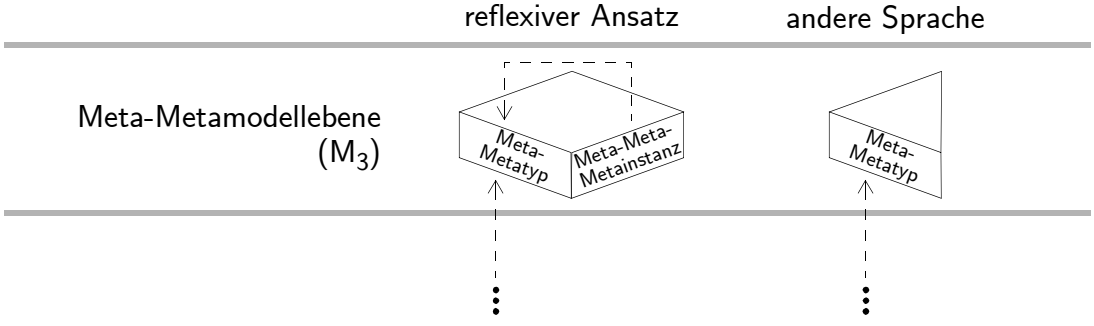


Abbildung 3-10. Alternativen zur Terminierung der Modellebenen

Ganz unproblematisch ist dieser Ansatz allerdings nicht, da er zu einem *Selbstreferenzierungsproblem* führt, auf welches wir im nächsten Abschnitt nochmals eingehen werden.

In unserem Grafikbeispiel ist die Modellierung der M_3-Ebene als reflexives Modell mit wenig Aufwand verbunden. Es müssen nur die zulässigen Typen für Attribute erweitert werden, da wir als Typ des Attributs multiplizitaet einen Objekttyp Multiplizitaet (und keinen Datentyp) definiert haben. In Abb. 3-11 ist das dahingehend erweiterte Modell gezeigt.

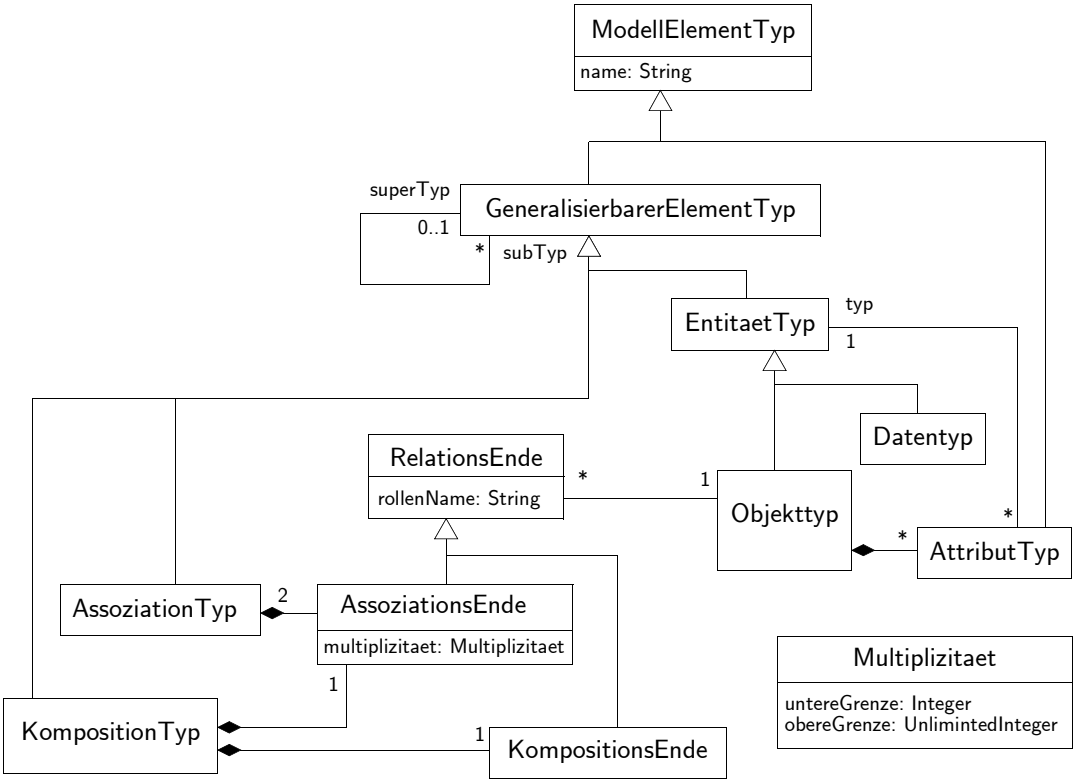


Abbildung 3-11. M_3-Ebene – Modell zur reflexiven Definition der Metaebene

Die Verallgemeinerung der Typen für Attribute erfolgt durch eine weitere Ebene in der Generalisierungshierarchie: EntitaetTyp beschreibt den Supertyp von Objekttyp und Datentyp. Da wir nur die Spezialisierung von Objekttypen erlauben wollen, müssen wir zur Beschränkung der Spezialisierung von Datentypen ein zusätzliches Constraint einführen, das die Definition von Supertypen für Datentypen verbietet:

$$\forall o@\mathsf{EntitätTyp} : |\mathsf{superTyp}(o)| > 0 \rightarrow o@\mathsf{Objekttyp}$$

Andere Sprache

Die zweite Lösung zur Terminierung der Modellebenen ist der Einsatz einer anderen Sprache zur Beschreibung der obersten Modellebene [Sei03], d.h. die Elemente der obersten Modellebene wären, wie in Abb. 3-10 dargestellt, nur Typen (und keine In-

stanzen eines anderen Metatyps). Neben einer natürlichsprachlichen (und damit i.d.R. nicht-formalen Beschreibung) existieren auch Vorschläge einer formalen Beschreibung der Modellkonzepte. In [ClE97] findet man z.B. die formale Definition der Kernkonzepte der objektorientierten Modellierung in der Sprache *Z*.

Der Nachteil dieser formal sauberen Lösung ist in der Praxis aber der Aufwand, den das Lernen zweier Sprachen mit sich bringt. Desweiteren verliert der „Meta"-Ansatz als Grundlage für eine Automatisierung oder Werkzeugunterstützung von Software-Entwicklungsaktivitäten an Mächtigkeit, da man die Elemente der obersten Ebene nicht wieder als Instanzen betrachten und dementsprechend darauf zugreifen kann (siehe dazu auch [Atk97]).

3.2.3 Diskussion der „traditionellen" Metamodellierung

Neben dem Vorteil der systematischen Beherrschung von Komplexität durch die Beschreibung einer Hierarchie von Metaebenen, die jeweils auf denselben grundlegenden OO-Konzepten basieren, gibt es bei der Metamodellierung leider auch Probleme. Die oben angedeutete Terminierung der Modellebenen ist nur eines davon. Ein weiteres Problem taucht auf, wenn man bei der in den vorangegangenen Abschnitten vorgestellten „traditionellen" Metamodellierung versucht, Konzepte über mehr als eine Metaebene hinweg zu beschreiben. Wir hatten bereits gesehen, dass die Modellierung des Attributs farbe für die Grafikelemente unseres Beispiels mit dem „traditionellen" Metamodellierungsansatz nicht zufriedenstellend durchgeführt werden kann.

Probleme des reflexiven Ansatzes

Zunächst wollen wir aber nochmals auf den reflexiven Ansatz zur Terminierung, wie er im letzten Abschnitt eingeführt wurde, eingehen. Wie bei jeder zirkulären Definition hat man hier das Problem, dass Information aus dem „Nichts" kommen muss, um eine solche Definition überhaupt zu verstehen [Atk97]. In einem formalen Kontext stellt dies ein kritisches Problem dar. Für die Modellierungspraxis lässt sich dieses Problem aber abschwächen, indem man demjenigen, der die zirkuläre Definition verstehen soll, zunächst einmal (nicht-formal) die wichtigsten Konzepte erklärt (also eigentlich – wie in der oben vorgestellten Alternativlösung – auf eine andere Sprache ausweicht). Auch in dieser Arbeit wurde dieser Weg gewählt, indem zunächst die wichtigsten Begriffe natürlichsprachlich geklärt und dann erst die Metamodelle eingeführt wurden. Hätte man das Kapitel in einer umgekehrten Reihenfolge (beginnend bei dem M_3-Modell) geschrieben, wäre ein Verständnis sicherlich schwierig gewesen.

Atkinson weist in [Atk97] auf ein interessantes Paradoxon bei einer reflexiven Definition der obersten Modellebene hin, das dazu führt, dass die Elemente der

obersten Modellebene auf jeder Modellebene existieren können. Nehmen wir an, M_t sei die oberste Modellebene, dann folgt per Definition, dass es sich bei der Instanziierung eines M_t-Modells um ein M_{t-1}-Modell handelt. Wegen der reflexiven Definition ist aber das M_t-Modell auch eine Instanziierung des M_t-Modells. Zusammen mit der ersten Feststellung folgt daraus, dass sich das M_t-Modell auf M_{t-1} befindet. Ausgehend von diesem Paradoxon argumentiert Atkinson, dass es – wie z.B. im Fall unseres Grafikbeispiels – nicht nötig wäre eine separate M_3-Ebene einzuführen, solange man auf einer beliebigen Ebene einen Satz von unveränderlichen Modellelementen zur Verfügung stellt. Da diese „Verschiebung" der reflexiven Ebene nichts an dem ursprünglichen Problem einer zirkulären Definition ändert, stellt in unseren Augen die Beschreibung der grundlegenden Modellbegriffe in einer separaten Modellebene die sauberere Trennung der Konzepte dar, an welcher wir im Verlauf dieser Arbeit festhalten wollen.

Aussagekraft eines Metamodells

Es soll an dieser Stelle darauf hingewiesen werden, dass die Klassifikation, welche durch ein Modell oder ein Metamodell beschrieben wird, wird im Allgemeinen nie eindeutig sein wird. Dies beginnt mit der Wahl der Namen für Objekttypen und wird mit deren Einordnung in mögliche Generalisierungs- und Instanziierungshierarchien oder durch die Beschreibung von Relationen nicht zweifelsfreier. Booch [Boo91, S.138] führt zum Beispiel ein Experiment an, in welchem zehn verschiedene Abbildungen von Zügen klassifiziert werden sollten. Die Probanden schlugen dabei 93 verschiedene Klassifikationen vor. In unserem Grafikbeispiel (siehe M_1-Modell in Abb. 3-4 auf Seite 42) hätte man z.B. den Objekttyp Haus auch direkt als Aggregation eines Dreiecks und zweier Rechtecke modellieren können, ohne den Objekttyp Etage zu modellieren.

Die Wahl einer sinnvollen und für eine Anwendung „richtigen" Modellierung hängt auch von den Anforderungen an das Modell (z.B. dynamische Erweiterbarkeit) und nicht zuletzt von einer technischen Realisierungsmöglichkeit (siehe Kapitel 5) ab. So führten wir zur Realisierung einer dynamischen Erweiterbarkeit eine Metaebene M_n ein, welche Elemente der Ebene M_{n-1} durch Instanziierung erzeugen kann. Umgekehrt haben wir aber auch gesehen, dass die Generalisierung sinnvoll sein kann, wenn man identische Eigenschaften (wie z.B. Attribute) für alle Subtypen an zentraler Stelle beschreiben möchte.

Wie man an dem Grafikbeispiel auch bereits gesehen hat, reicht die bloße grafische Darstellung der Modelle nicht für ein vollständiges Verständnis aus, denn diese beschreibt lediglich die abstrakte Syntax (siehe Abschnitt 2.1 auf Seite 9 und [Mor99]). Etwas mächtiger werden diese Diagramme durch die Beschreibung von Constraints (zu denen auch die Kardinalitäts-Constraints, die durch Multiplizitäten beschrieben werden, gehören). Mit Hilfe von Constraints lassen sich die geforderten

„Well-formedness"-Regeln (statische Semantik) spezifizieren. Darüber hinaus ist aber auch die Bedeutung, also die (dynamische) Semantik, der modellierten Konzepte relevant. Ohne zu wissen, was ein Grafikelement ist, würde man das Modell aus Abb. 3-4 wohl nur schwer verstehen. Ein gutes (Meta-)Modell muss also eine präzise Beschreibung des Sachverhalts erlauben, weil es sonst wertlos wird.

Bisher haben wir die Semantik der Modelle größtenteils nicht-formal (durch natürlichsprachlichen Text) beschrieben, was auch für den Verlauf der Arbeit genügen wird, da die Automatisierungswerkzeuge nur syntaktische Modelltransformationen vornehmen (siehe Abschnitt 2.1.1 auf Seite 11). Einige der Aspekte der Modelle, so z.B. die Instanziierung, Generalisierung oder Attributbelegung, haben wir jedoch bereits formalisiert, indem wir diese Begriffe mengentheoretisch aufgefasst haben. Mögliche Ansätze zu einer vertiefenden und präzisen Definition der dynamischen Semantik von OO-Modellen findet man z.B. in [LCA02] und [ClE97].

Probleme der „flachen" Instanziierung

Bei unserer bisherigen Vorstellung der Metamodellierung sind wir stets von einer „flachen" Instanziierung ausgegangen. Dies bedeutet, dass Attribute oder Assoziationen der Ebene M_n auf der Ebene M_{n-1} *immer* als Attributinstanzen (Attributbelegungen) und Links auftauchen. Demzufolge sind sie auch nicht weiter instanziierbar. Diese Tatsache war z.B. mit ein Grund dafür, dass wir die Assoziation zwischen einem Grafikelement und seinen Punkten auf der darüberliegenden Modellebene als Objekttyp modellieren mussten.

Atkinson und Kühne führen in [AtK01] die Ursache dieses Problem auf den Grundansatz zurück, dass ein Typ immer nur seine *direkten* Instanzen beschreiben kann und keinen Einfluss auf die Instanziierung auf weiter unten liegenden Ebenen haben kann. Einzig Objekte (Instanzen von Objekttypen) hatten die Eigenschaft, dass sie wiederum als Objekttypen interpretiert werden konnten. Obige Tatsache ist die Ursache für ein Problem, dass die Autoren „Replikation von Konzepten" nennen. In gewisser Weise stellten wir bei der Beschreibung der unterschiedlichen Ebene ja bereits fest, dass sich einige der Konzepte (z.B. die Modellierung von Assoziationen) ähneln oder sogar gleichen.

Ein Ansatz, der zumindest eines der Probleme vermindert, ist der Einsatz sog. „Powertypes" [Ode98, S.23ff.]. Wir hatten bei dem Ziel ein farbe-Attribut für alle Grafikelemente zu definieren (siehe Abschnitt 3.2.1) bereits bemerkt, dass dies nur durch die Einführung eines Supertyps GrafikElement auf M_1 möglich war. Damit müssen wir uns aber an die Konvention halten, alle Metainstanzen von GrafikElementTyp zusätzlich als Subtyp von GrafikElement zu definieren. Powertypes bieten nun die Möglichkeit diesen Sachverhalt zu beschreiben, denn ein Powertype ist ein „[Meta-]Objekttyp, dessen Instanzen Subtypen eines anderen Objekttyps sind" [Ode98, S.28] und dieser „andere" Objekttyp lässt sich bei der Definition eines Po-

wertypes entsprechend angeben. Auch die UML kennt Powertypes (siehe z.B. RJB99, S.392]) und stellt damit eine Möglichkeit zur Verfügung, sowohl die Instanz- als auch die Typfacette eines Elements zu beschreiben.

Leider ist der Powertype-Ansatz nicht unproblematisch. Kühne und Steimann [KüS04] identifizieren drei Probleme:

1. Zur Definition eines Powertypes muss sowohl der Typ als auch der Metatyp, dessen Instanzen ein Subtyp des Typs werden sollen, angegeben werden. Daher überschreitet eine dieser Abhängigkeitsbeziehungen die Ebenengrenzen, was Komplikationen bzgl. der sauberen Trennung der Ebenen (wie wir sie ja bei der strikten Metamodellierung eingeführt hatten) darstellt.
2. Die Beschreibung der Typ- und Instanzfacetten ist auf zwei Stellen (Typ und Metatyp) verteilt.
3. Bei mehr als drei Ebenen skaliert das Powertype-Konzept schlecht, wenn man die Instanziierung über mehr als eine „Zwischenebene“ hinweg beschreiben möchte.

Daher ist der „traditionelle“ Ansatz trotz der Verfügbarkeit von Powertypes nicht zufriedenstellend.

Neben den angesprochenen Problemen lässt zusätzlich auch die Lesbarkeit von Metamodellen in diesem Ansatz zu wünschen übrig, da die Modelle eine unnötige Komplexität aufweisen. Diese Unleserlichkeit der Modelle ist sicherlich auch darin begründet, dass man wieder nur *außerhalb* des Modells festlegen kann, wie die konkrete Notation für Instanzen eines M_n-Typs aussehen soll. So muss man z.B. im obigen Grafikbeispiel festlegen, dass aus dem M_n-Objekttyp AssoziationTyp eine M_{n-1}-Relation wird.

Als Basis für diese Arbeit wollen wir daher im folgenden Kapitel einen verbesserten Ansatz zur Multiebenenmodellierung einführen, der die oben aufgeführten Probleme beseitigt.

Zusammenfassung

In diesem Kapitel wurde zunächst eine konsistente Definition der wichtigsten objektorientierten Begriffe vorgenommen. Ausgehend von dieser Begriffsbasis konnte dann das Konzept der Metamodellierung eingeführt und an Hand eines durchgängigen Beispiels vorgeführt werden. Kritische Probleme der „traditionellen“ Metamodellierung wurden diskutiert, unter anderem die Terminierung der Modellebenen und die „flache“ Instanziierung.

4 Ein verbesserter Ansatz zur modellbasierten Automatisierung

Aluminium, Al, chem. Element aus der III. Hauptgruppe des Periodensystems, der Gruppe der Erdmetalle, das wichtigste Leichtmetall, ein Reinelement.

— [Fachlexikon ABC Chemie, 1976]

Das vorangegangene Kapitel legte die Grundlagen für eine objektorientierte Modellierung und Metamodellierung. Allerdings kristallisierten sich bei der in der Praxis üblicherweise eingesetzten Metamodellierungstechnik Probleme heraus. In diesem Kapitel wird daher eine Verbesserung dieses „traditionellen“ Ansatzes vorgestellt und auf der Basis dieser verbesserten Multiebenenmodellierung die Sprache AL++ zur operationalen Beschreibung von Modelltransformationen entwickelt.

4.1 Verbesserte Multiebenenmodellierung

Als Lösung der in Abschnitt 3.2 auf Seite 38 identifizierten Probleme wird in diesem Abschnitt eine verbesserte Multiebenenmodellierung vorgeschlagen, welche die Erstellung und die Handhabung von Metamodellen deutlich vereinfacht.

4.1.1 Das Grundkonzept der tiefen Instanziierung

Atkinson, Kühne und Steimann stellen in [AtK01] und [KüS04] die *tiefe Instanziierung* als Lösungsansatz für die im vorangegangenen Kapitel erläuterten Probleme vor. Die Grundidee der tiefen Instanziierung ist, die instanziierbaren Elemente einer jeden Modellebene an Hand von vom *Modellierer* vorgegebener Eigenschaften ableiten zu können. Im Gegensatz dazu muss beim „traditionellen“ Ansatz die im Modell

nicht sichtbare *Vereinbarung* getroffen werden, ob ein Element der Ebene M_n erneut zu einer Instanz der Ebene M_{n-1} instanziiert werden kann. So hatten wir in Abschnitt 3.2.3 auf Seite 51 bereits erwähnt, dass bei der „traditionellen" Metamodellierung z.B. nur Objekte wieder als Typen von Instanzen der darunterliegenden Ebene interpretiert werden können.

Zur Beschreibung der möglichen Instanziierungstiefe führen die Autoren daher ein Merkmal für alle Modellierungselemente ein, das sie *Potenz* nennen. Die Potenz eines Modellelements gibt dabei an, bis zu welcher Tiefe das Element instanziiert werden kann, wobei jede Instanziierung diesen Wert erniedrigt. Ein Element der Potenz 0 entspricht damit einer Instanz, die nicht weiter instanziiert werden kann (also z.B. einer Attributbelegung oder einem Link).

Zusammen mit der Nummer der Modellebene, die wir bisher ja schon durch eine Zahl kennzeichneten, erlaubt dieser Ansatz die eindeutige Beschreibung der jeweiligen Typ- und Instanzeigenschaften jedes einzelnen Modellelements. Da z.B. jeder GrafikElementTyp (definiert auf Ebene M_2, siehe Abb. 3-7 auf Seite 44) zu GrafikElement instanziiert werden kann, der Instanzen auf M_0 besitzt, hätte dieser eine Potenz von 2. Die Generalisierungsrelation hingegen hätte eine Potenz von 0 (sie existiert nur auf der Ebene, auf der sie definiert wird). Als Notation schlagen die Autoren vor:

$$\text{Modellelement}_n^p\,,$$

wobei p der Potenz und n der Ebene entspricht (den Index zur Beschreibung der Modellebene hatten wir bisher schon genutzt). Für unser Beispiel würden wir dann z.B. schreiben

$$\text{GrafikElementTyp}_2^2\,,$$

wenn wir den Objekttyp GrafikElementTyp der Ebene M_2 meinen.

Neben diesem Grundkonzept führen die Autoren eine weitere Unterscheidung bei der Instanziierung von Attributen ein. Dabei wird zwischen „einfachen" Attributen, die nur bei einer Potenz von 0 eine Wertbelegung erfahren, und „dualen" Attributen, denen auf jeder Instanziierungsbene ein Wert zugewiesen werden kann, unterschieden. Die Semantik eines „dualen" Attributs der Potenz p führen Atkinson und Kühne dabei auf eine Menge von „einfachen" Attributen mit den unterschiedlichen Potenzen $(p, p-1, \ldots, 1, 0)$ zurück. Wegen dieser Äquivalenz werden wir in den Fällen, wo „duale" Attribute vorkommen, diese explizit als eine Menge von „einfachen" Attributen modellieren, was eine Reduktion der verwendeten Begriffe ermöglicht.

Mit der in Gl. 3-1 auf Seite 30 eingeführten Notation ergibt sich dann folgende Eigenschaft der Instanziierung:

$$e_{n-1}^{p'} @ T_n^p \text{ mit } p' < p\,. \qquad \textbf{(Gleichung 4-1)}$$

Zur Vereinfachung der Darstellung können diejenigen Parameter weggelassen werden, die eindeutig sind, so z.B. die Ebene, wenn man ein Modell einer einzelnen Ebene beschreibt. Fehlen die Angaben zur Potenz, dann werden die Modellelemente in ihrer „traditionellen" Bedeutung interpretiert (also Attribut und Assoziation mit Potenz 1, Generalisierung mit Potenz 0, usw.).

4.1.2 Verallgemeinerung und Erweiterung des Ansatzes

Wendet man das Konzept der tiefen Instanziierung auf unterschiedliche Modellentitäten an, so stellt sich heraus, dass die Angabe einer Potenz auch Auswirkungen auf die Art der Instanziierung solcher Entitäten hat. Zum einen kann eine solche Instanziierung explizit (oder „manuell") erfolgen, zum anderen implizit (oder „automatisch"). In Tabelle 4-1 werden die möglichen Ausprägungen zunächst gegenübergestellt.

Tabelle 4-1. Art der Instanziierung bei höheren Potenzen

Modellentität	**explizite Instanziierung**	**implizite Instanziierung**
Objekttyp	möglich (Normalfall)	nicht möglich
Attributtyp	möglich	möglich
Relationstyp	möglich	nicht möglich
Operationstyp	möglich (hier nicht behandelt)	möglich (mit Einschränkungen)

Dass eine explizite Instanziierung von Objekttypen möglich ist, erscheint offensichtlich, handelt es sich hierbei doch um den Normalfall. Eine automatische Instanziierung ist hingegen jedoch nicht zu erzielen, da das Modell auf der Metaebene nicht genügend Informationen für eine solche Instanziierung beinhaltet. Insbesondere fehlt der Name der Objekttyp-Instanz.

Demgegenüber kann man durch die Angabe eines Attributnamens auf einer höheren Modellebene dessen automatische Definition bei der Instanziierung des zugehörigen Objekttyps erreichen. So erscheint ein Attribut a_n^p eines Objekttyps OT_n^q automatisch als Attribut a_{n-1}^{p-1} aller Objekte o mit

$$o_{n-1}^{q'} @ OT_n^q\,,\ q' < q\,. \qquad \textbf{(Gleichung 4-2)}$$

Alle diese „Attributinstanzen" besitzen folglicherweise denselben Namen a.

Dieser Automatismus, der von Atkinson et al. beschrieben wird (siehe oben), stellt in meinen Augen allerdings nur *eine* mögliche Art der Instanziierung von At-

tributen dar. Ebenso können diese durch die explizite Instanziierung eines Attributtyps – unter Angabe des gewünschten Attributnamens – erzeugt werden.

Um diese beiden Arten der Instanziierung von Attributen in den Modellen unterscheiden zu können, wollen wir (in Abweichung von Atkinson et al.), die automatisch instanziierten Attribute durch Unterstreichungen kenntlich machen. Da Attribute der Potenz eins – die „normalen Attribute“ – immer automatisch instanziiert werden, werden auch diese unterstrichen (in Abweichung von der UML-Syntax, wo die Unterstreichung für Instanzen vorgesehen ist).

Mit einer Definition eines höherpotenten Relationstyps lässt sich wiederum kein Automatismus verbinden, da hier – wie bei den Objekttypen – die entsprechende Information auf der Metaebene fehlt. Ein solcher höherpotenter Relationstyp stellt also lediglich eine Klassifikation möglicher Elemente der *darunterliegenden* Ebene dar.

Wichtig ist diese Tatsache insbesondere bei der Betrachtung der Kardinalitäts-Constraints (Multiplizitäten). Die auf einer Ebene M_n angegebenen Multiplizitäten (und auch Rollennamen) gelten immer nur für die Instanziierung der Relation auf der darunterliegenden Ebene M_{n-1} – geben also *nicht* die Kardinalitäts-Constraints für den Link der Potenz 0 am Ende der Instanziierungskette an. Umgekehrtes gilt für die Art der Relation (Generalisierung, Aggregation, Komposition). Hier trifft die beschriebene Art der Relation nur für das Element mit einer Potenz $p \leq 1$ zu.

Zuletzt lässt sich natürlich auch für Operationen das Konzept der tiefen Instanziierung anwenden. Normalerweise besitzen die Operationsdefinitionen eine Potenz von „1“, was die Existenz von Operationsinstanzen auf der darunterliegenden Modellebene impliziert. Definiert man eine Operation mit einer Potenz größer eins, so wird diese erst auf einer weiter unten liegenden Ebene instanziiert und kann dann dementsprechend aufgerufen werden.

Eine automatische Instanziierung (gekennzeichnet durch eine Unterstreichung des Operationsnamens) ist auch bei Operationen nur möglich, falls auf der Metaebene genügend Informationen vorhanden sind. Dieses ist dann der Fall, wenn in der Definition der Operation nur automatisch instanziierte Attribute verwendet werden, die höchstens dieselbe Potenz wie die Operation aufweisen. Zusammen mit der tiefen Instanziierung von Attributen bietet diese Definition von Operationen eine Alternative zu der Definition von Supertypen, welche Attribute und Methoden besitzen, die aus den Subtypen herausfaktorisiert wurden (siehe dazu Abschnitt 3.2.1 auf Seite 40).

Eine explizite Instanziierung wäre denkbar, wenn bei der Instanziierung die fehlende Information angegeben würde. Dies wäre möglich, wenn bei der Operationsdefinition entsprechende Parameter definiert würden. Auf diesen Aspekt wollen wir in dieser Arbeit allerdings nicht näher eingehen.

4.1.3 Ein Beispiel zur verbesserten Multiebenenmodellierung

Zur Illustration der verbesserten Multiebenenmodellierung wollen hier nochmals unser Grafikbeispiel aufrollen und an Hand dieses Beispiels zeigen, wie durch die Anwendung der tiefen Instanziierung und unserer Ergänzungen die Metamodelle vereinfacht werden können und auch die Definition des gewünschten farbe-Attributs (vgl. Abschnitt 3.2.1 auf Seite 40) möglich wird.

Um die Bedeutung der Potenzen besser illustrieren zu können beginnen wir – anders als bei der ursprünglichen Vorstellung des Beispiels in Abschnitt 3.2.1 auf Seite 40 – mit der Erklärung der Metamodellebene M_2 (siehe Abb. 4-1). In Abb. 4-2 ist eine mögliche Instanziierung dieser Ebene (M_1) gezeigt.

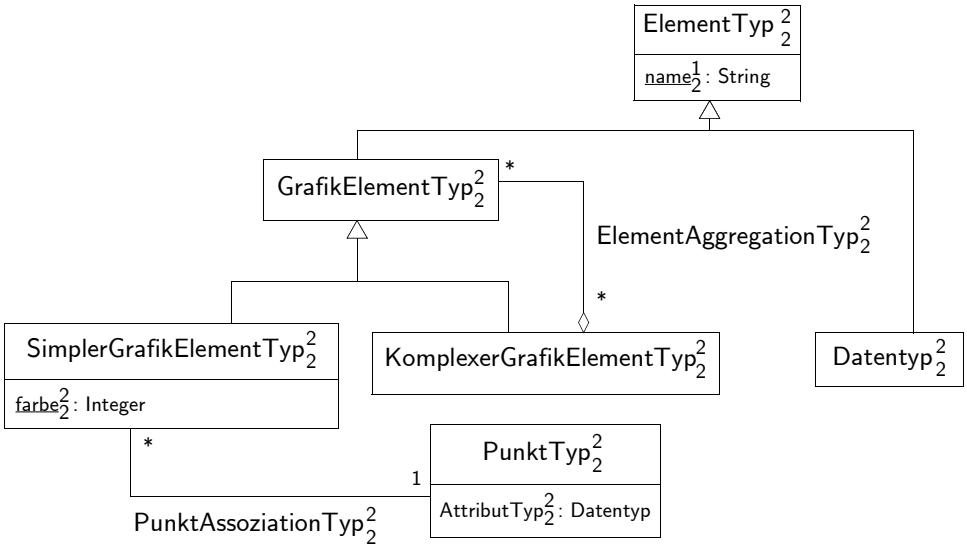


Abbildung 4-1. M_2-Ebene des Grafikbeispiels in verbesserter Multiebenenmodellierung

In beiden Modellen wurden die Potenzen und die Ebenennummern für alle Modellelemente angegeben, um das Konzept der Potenz auch für „traditionelle“ Modellelemente zu verdeutlichen.

Im Vergleich zu dem M_2-Modell aus Abb. 3-7 auf Seite 44 stellt man fest, dass die Modellierung von Beziehungstypen nun sehr viel übersichtlicher durch eine Definition von Potenzen größer eins erfolgt. So ist der PunktAssoziationTyp und der ElementAggregationTyp nun kein Objekttyp sondern eine Relation mit einer Potenz von zwei. Diese Typen führen demzufolge zu Instanzen der Potenz eins, also zu „normalen“ Relationen.

Auch die Definition des Attributtyps AttributTyp im Objekttyp PunktTyp stellt eine deutliche Vereinfachung gegenüber der ursprünglichen Modellierung über einen separaten Objekttyp mit Kompositionsbeziehung dar.

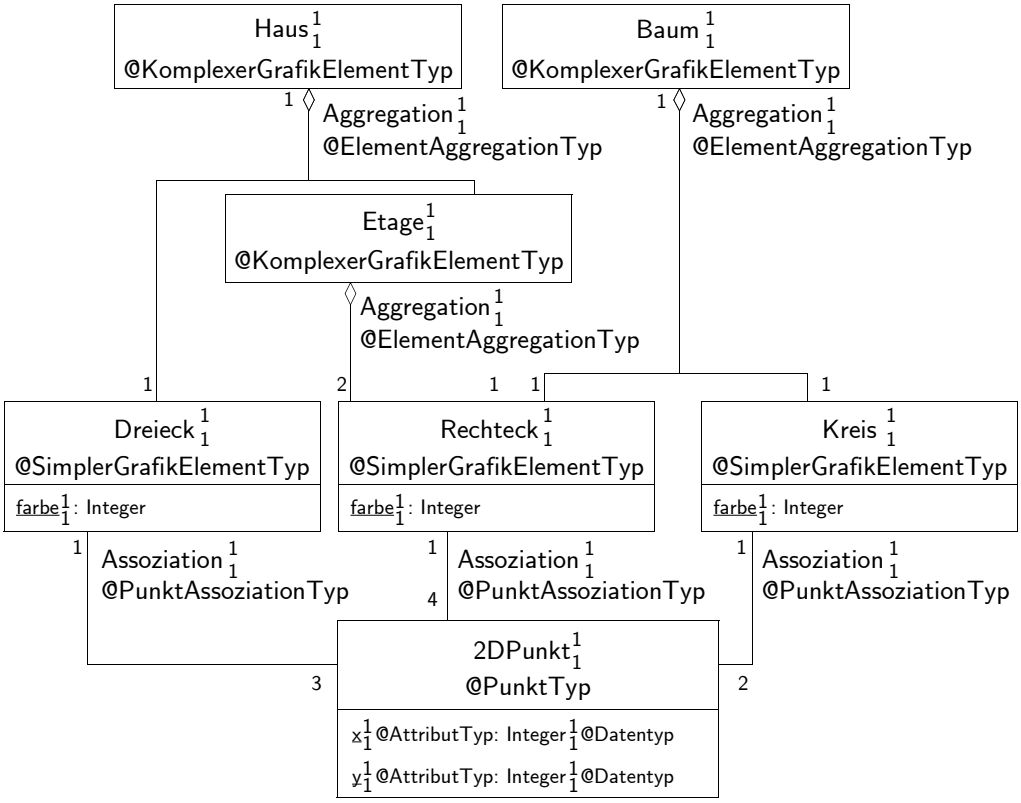


Abbildung 4-2. Modellebene (M_1) des Grafikbeispiels mit tiefer Instanziierung

Durch die Definition eines Attributs farbe der Potenz zwei für den Objekttyp SimplerGrafikElementTyp erreichen wir nun auch ohne die Einführung einer Supertyps SimplesGrafikElement die automatische Definition eines Attributs farbe (der Potenz eins) für jede M_1-Instanz von SimplerGrafikElementTyp.

Für eine mögliche Darstellung der reflexiven M_3-Ebene verweisen wir an dieser Stelle auf die Arbeit von Atkinson und Kühne [AtK01] bzw. auf Abschnitt 11.2.1 auf Seite 343. Kernpunkt des von den Autoren „Metamodel for Multiple-Metalevels“ genannten Modells ist der Meta-Metatyp ModelElement, welcher die Attribute level und potency besitzt.

4.1.4 Fazit

Metamodellierung wird auch in der industriellen Praxis immer populärer. Dies liegt nicht zuletzt an der MDA-Initiative der OMG (siehe Abschnitt 5.3.2 auf Seite 120), in welcher Modelle die primären Entwicklungsartefakte darstellen, und an dem Versuch, die neue UML 2 auf einer soliden Metamodellinfrastruktur aufzubauen (siehe [BHK04]). Umso mehr ist die Tatsache zu bedauern, dass noch keine Einigung über die grundlegenden Konzepte der OO-Modellierung besteht (wir hatten an der ein oder anderen Stelle auf die entsprechenden Diskussionen hingewiesen) oder viele der

Begriffe austauschbar verwendet werden. Dies mag in einem speziellen Kontext nicht problematisch sein, ist für das genaue Verständnis (und dann auch Anwendung) einzelner Konzepte aber hinderlich. Als Beispiel seien die Begriffspaare „Objekttyp“/„Klasse“ und „Spezialisierung“/„Vererbung“ angeführt. In Abschnitt 3.1.1 auf Seite 31 hatten wir die genauen Unterschiede erläutert.

Auch die Tatsache, dass sich fortschrittliche und notwendige Konzepte der Metamodellierung, wie die vorgestellte tiefe Instanziierung, noch nicht etablierten ist insofern zu bedauern, als dass man für diese Konzepte auf eine Werkzeugunterstützung verzichten muss und auch auf keine gemeinsame (und einfache) „Sprache“ zwischen Modellierern zurückgreifen kann. Auch wenn wir in dieser Arbeit die verbesserte Multiebenenmodellierung als Technik zur Beschreibung unserer Modelle einsetzen werden, müssen wir an den Stellen, an denen eine direkte Abbildung auf die Automatisierungswerkzeuge nicht möglich ist, auf die „traditionelle“ Modellierung ausweichen, bzw. die fortschrittlichere Darstellung auf die „traditionelle“ Darstellung abbilden. Dass sich beide Darstellungen nutzen lassen, hatten wir am Grafikbeispiel bereits durchgespielt.

4.2 Modelltransformation mit Metamodellen

In den vorangegangenen Abschnitten und in Kapitel 3 wurden die Grundlagen einer Beschreibung von Modellen auf verschiedenen Abstraktionsebenen gelegt. Dabei beschreibt z. B. eine Instanz eines M_2-Modells ein Entwicklungsmodell (Software-Spezifikation) zu einem Zeitpunkt der Entwicklung. Eine Aktivität in der Software-Entwicklung kann man daher als eine Abbildung zwischen solchen Modellinstanzen verstehen. Diese Transformationen, welche durch ein Prozessmodell (siehe dazu Abschnitt 2.2.2 auf Seite 19) spezifiziert werden, wollen wir nun so beschreiben, dass deren Automatisierung möglich wird.

Betrachtet man allgemein eine Abbildung zwischen Modellen, so lässt sich eine solche Transformation aus zwei Blickwinkeln betrachten: der *Beschreibung* und der *Ausführung*. Die Beschreibung einer Transformation erfolgt durch das Prozessmodell unter Bezug auf Elemente des Produktmodells. Dies impliziert, dass eine solche Transformation auf der Ebene der Produktmodellelemente (oder evtl. höher) spezifiziert werden muss. Betrachtet man eine Transformation eines M_n-Modells, so muss diese Abbildung also auf einer Ebene M_m mit $m > n$ spezifiziert werden. Die Ausführung der Transformation findet dann auf der Ebene M_n statt.

Obigen Sachverhalt kann man nun auch so interpretieren, dass eine auf Ebene M_m spezifizierte Transformation das *Verhalten* der M_{m-1}-Modelle angibt. Aus diesem Grund können wir die verhaltensbeschreibenden OO-Begriffe, die wir zu Beginn von Kapitel 3 eingeführt hatten, nun auch zur Spezifikation von Modelltransformationen einsetzen.

4.2.1 Spezifikation von Modelltransformationen

Zwei konträre Ansätze zur Spezifikation des Verhaltens und damit zur Beschreibung der Modelltransformationen sind typischerweise möglich: ein *imperativer* und ein *deklarativer Ansatz*.

Beim deklarativen Ansatz werden Transformationen durch Regeln beschrieben, die jeweils durch Vor- und Nachbedingungen (also Constraints, siehe Abb. 3.1.3 auf Seite 37) spezifiziert werden. Dabei beschreibt die Vorbedingung den „System"-Zustand (siehe Abschnitt 3.1.2 auf Seite 35) *vor* Ausführung der Transformation und die Nachbedingung den Zustand, der *nach* der Transformation erreicht wird. Beim imperativen Ansatz dagegen werden die Aktivitäten beschrieben, mit Hilfe derer man vom anfänglichen Systemzustand zum Zustand nach der Transformation gelangt [Wie98, S. 480].

Da es sich bei Metamodellinstanzen um Objektgraphen handelt, findet man als Vertreter des deklarativen Ansatzes sehr häufig die *Graphtransformation*, welche auch *Graphreduktion* oder „*Graph-Rewriting*" genannt wird. Nach [AEH99] besteht eine Regel r einer Graphtransformation aus einer Vorbedingung L, „*Left-Hand Side*" (LHS) genannt, und einer Nachbedingung R, die als „*Right-Hand Side*" (RHS) bezeichnet wird. Die Anwendung der Regel r auf einen Graph G ersetzt ein Vorkommen von L durch R. Insofern entspricht die Regel einer solchen Graphtransformation einer Regel in einer Graph-Grammatik (siehe Definition 2-5 auf Seite 15).

Die Anwendung einer Regel erfolgt im allgemeinsten Fall durch das Entfernen des Vorkommens von L aus G, was zu einem Graphen D führt. Anschließend wird R in den Graphen D eingefügt und dabei die Verbindung von R zu dem Graphen D durch Einführung neuer Kanten sichergestellt. Beide Operationen werden in der Regel r spezifiziert. In Abschnitt 2.1.2 auf Seite 13 hatten wir bereits auf das Einführen von sog. Kontextelementen [ReS97] zur Realisierung dieser Einbettung hingewiesen.

Schließlich kann eine Regel auch eine weitere Vorbedingung beinhalten, die eine Regelanwendung weiter einschränkt. Czernecki und Helsen präsentieren in [CzH03] eine genauere Klassifikation solcher Graphtransformationsansätze.

Die deklarative Beschreibung ist sehr gut geeignet für abstrakte oder unvollständige Transformationen [BFJ03], da sie von einer möglichen Implementierung oder Realisierung abstrahiert [MTA99]. Daher ist eine solche Art der Spezifikation in frühen Stadien der Software-Entwicklung angezeigt, da Entwicklungsentscheidungen, die für eine Realisierung der Transformation notwendig sind, noch nicht getroffen wurden [ACR03] oder noch nicht getroffen werden sollen, um die Freiheitsgrade der Entwickler nicht einzuschränken. Desweiteren bietet eine deklarative Spezifikation den Vorteil, dass sie einer formalen Verifikation leichter zugänglich ist. Varro et al. [VGP01] schlagen z. B. vor, die Korrektheit der Transformationsregeln sicherzustel-

len, was eine Korrektheit der Ausgabemodelle impliziert. Bei einem metamodellbasierten Ansatz muss man insbesondere fordern, dass das Ausgangsmodell eine gültige Instanz des Metamodells (Produktmodells) ist.

Bei Graphtransformationen kommt es allerdings häufig zu Nicht-Determinismen [Por02, S.4][GLR02]. Dies liegt daran, dass man zunächst eine der möglichen Regeln, die anwendbar sind, auswählen muss. Desweiteren kann eine Regel oft auf mehrere Vorkommen der LHS angewendet werden. Daher hängt das Ergebnis einer Graphtransformation von diesen Entscheidungen ab, die – wenn keine weiteren Einschränkungen spezifiziert werden – völlig beliebig getroffen werden können (vgl. [AEH99]). Weiter muss man ein Graphtransformationssystem sehr sorgfältig konstruieren, da man ansonsten die Terminierung der Graphersetzung nicht garantieren kann (siehe dazu [AEH99, S.9]). Würde z.B. die Anwendung einer Regel immer zu einem Ausgabegraphen führen, der die Anwendung des LHS-Ausdrucks der Regel erlaubt, könnte solch ein Graph nie reduziert werden. Rekers und Schürr schlagen in [ReS97, S.4] als Lösung des Problems solcher zyklischen Ableitungen zum Beispiel vor, die Regeln so zu wählen, dass die LHS lexikographisch kleiner als die RHS ist.

Vor einer tatsächlichen Ausführung einer deklarativen Transformation muss diese zunächst in eine operationale Form überführt werden. Dazu gehört zum einen die Auswahl eines (deterministischen) Algorithmus, der das „Pattern-Matching“ der LHS-Ausdrücke vornimmt und die oben beschriebenen Modelländerungen realisiert. Um die herausgestellen Vorteile der Verifizierbarkeit nicht zu verlieren, muss man dazu also eine automatische Abbildung der Transfromationen auf Code vornehmen („Correctness by Construction“). Bei komplexen Graphtransformation führt dies leider häufig zu ineffizientem Code, da das effiziente Matching von LHS-Ausdrücken problematisch ist [AEH99][BFJ03] und auch die Wahl eines effizienten Algorithmus durch die reine Angabe von Vor- und Nachbedingung nicht garantiert ist [MTA99].

Im Gegensatz zu den deklarativen Ansätzen werden – wie oben erwähnt – bei einem imperativen Ansatz die Modelltransformationen direkt durch Operationen beschrieben. Diese Operationen (siehe Abschnitt 3.1.2 auf Seite 35) werden dabei imperativ (auch operational) innerhalb der im Modell festgehaltenen Objekttypen spezifiziert.

Dieser Ansatz bietet im Vergleich zur deklarativen Beschreibung den pragmatischeren und auch effizienteren Weg. Zum einen entfällt die notwendige Abbildung der deklarativen Beschreibung auf eine operationale Darstellung. Zum anderen ergeben sich keine Indeterminismen, da die Ausführung der operationalen Beschreibung normalerweise deterministisch abläuft. Auch die Effizienz ist deutlich besser als bei den deklarativen Ansätzen, da das aufwändige Suchen von LHS-Instanzen im Eingabemodell entfallen kann.

Schließlich lassen sich auch „anspruchsvollere“ Transformationen beschreiben, welche z.B. mit temporären Datenstrukturen oder geschachtelten Rekursionen arbeiten. Wegen der Lokalität [AEH99] der Graphersetzungsregeln (es wird immer *eine* Instanz der LHS durch eine RHS ersetzt) ist dies um vieles aufwändiger zu beschreiben, insbesondere auch was die Erzeugung und Veränderung komplexer temporärer Datenstrukturen angeht. Zuletzt erlaubt der operationale Ansatz die Wiederverwendung hinlänglich bekannter Algorithmen (z.B. zur Tiefensuche in Bäumen).

Zur Übersicht sind die beiden Ansätze in der folgenden Tabelle 4-2 nochmals gegenübergestellt.

Tabelle 4-2. Vergleich des deklarativen und imperativen Ansatzes zur Beschreibung von Modelltransformationen

Aspekt	**deklarativer Ansatz**	**imperativer Ansatz**
Realisierungstechnik	Graphtransformation, Spezifikation logischer Ausdrücke (OCL), Transformationssprache (XSLT) [GLR02]	Aktionssprache (UML/Action-Semantics) [SPH01], Programmiersprache, Skriptsprache [Por02]
Abstraktion	realisierungsunabhängig: abstrakte/ unvollständige Transformationen möglich [BFJ03]	realisierungsabhängig, evtl. implementierungsunabhängig (Aktionssprache)
Ausführbarkeit	Operationalisierung der Regeln notwendig	direkt möglich
Mächtigkeit	Lokalität von Graphtransformationsregeln kann einschränken [AEH99]	komplexe Änderungen, Rekursion und temporäre Datenstrukturen realisierbar
Vorhersagbarkeit	bei Graphtransformation: Nicht-Determinismen und Nicht-Terminierung möglich [AEH99][GLR02]	i.d.R. deterministisch
Effizienz	Wahl eines effizienten Algorithmus aus Regeln oft nicht garantiert [MTA99]	Einsatz bekannter (effizienter) Algorithmen möglich

Wie man sieht, fällt die Wahl eines entsprechenden Ansatzes nicht leicht und sollte daher in Abhängigkeit von der konkreten Anwendung getroffen werden. So kann eine template-basierte Sprache zur Code-Erzeugung sehr bequem sein. Zur einfachen Abfrage von Modellinformationen (ohne eine Transformation durchzuführen oder intermediäre Datenstrukturen zu verwenden) sind logische Ausdrücke (z.B. in der OCL, siehe Abschnitt 8.1.3 auf Seite 194) sehr geeignet, da sie einfach zu spezifizieren sind.

Auch die Object Management Group hat die Notwendigkeit eines passenden Mechanismus für die Transformation von Modellen erkannt und daher ein „Request for Proposals“ zu „Queries/Views/Transformations“ formuliert. Der Beitrag der QVT-Partners zu diesem „Request“ (siehe [ACR03]) sieht so z.B. explizit die Möglichkeit

der Spezifikation (deklarativ) und Implementierung (operational) von Transformationen vor.

Ein Vergleich verschiedener Ansätze zur Modelltransformation wird auch von Gerber et al. in [GLR02] vorgestellt. Die Autoren favorisieren darin den deklarativen Ansatz wegen der leichteren Verständlichkeit der Regeln, erkennen aber auch die Notwendigkeit imperativer Aspekte für manche Aktivitäten an (wie z.B. Aktualisierung von Modellen). Sendall et al. heben hervor, dass Entwicklern in der Regel die Beschreibung komplizierter Algorithmen (Transformationen) in einer prozeduralen Sprache leichter fällt [SeK03, S.43]. Gardner et al. [GGK03, S.192] gehen noch einen Schritt weiter und behaupten, dass für die Anwender von Modelltransformationen die Beschreibung komplexer Transformationen alleine mit dem deklarativen Ansatz nicht möglich sei, und schlagen daher für solche Fälle vor, einer imperativen Realisierung den Vorzug zu geben.

Die letzte Aussage deckt sich auch mit unseren Beobachtungen und Erfahrungen. Für unsere Anwendung hat sich der imperative Ansatz insbesondere daher bewährt, da wir viele komplexe Abfragen (Kapitel 8) und Modelltransformationen mit globalen Abhängigkeiten (Kapitel 9) automatisierten. Daher werden wir in dieser Arbeit hauptsächlich eine operationalen Beschreibung von Entwicklungsaktivitäten verwenden.

4.2.2 Die Aktionssprache AL++

Zur operationalen Verhaltensbeschreibung (und damit für die Modelltransformation) können verschiedene Techniken eingesetzt werden. Eine Technik ist die Beschreibung des Verhaltens durch Endliche Automaten. In Abschnitt 3.1.2 auf Seite 35 hatten wir diese bereits erwähnt. Endliche Automaten sind sehr gut geeignet für eine Verhaltensbeschreibung von Systemen, die stark ereignisgesteuert ablaufen. Für eine Transformation von Modellen ist diese Technik nicht sonderlich geeignet. Sinnvoller ist da eine *algorithmische* Beschreibung des Verhaltens, wie man es z.B. von imperativen Programmiersprachen kennt. Zur implementierungsunabhängigen Beschreibung solcher Algorithmen bieten sich die sogenannten *Aktionssprachen* („*Action Languages*“) an [MTA99].

Die objektorientierte Modellierungssprache UML besitzt ab der Version 1.5 [OMG03] z.B. eine solche Aktionssprache. Die OMG spezifiziert für diese *UML Action Semantics* [SGJ02] genannte Sprache jedoch nur die abstrakte Syntax, welche durch ein Metamodell beschrieben wird. Bezüglich der konkreten Syntax („*Surface Notation*“ genannt) macht die Spezifikation keine Vorgaben. Allerdings wird die Abbildung existierender Aktionssprachen wie z.B. der freien Action Specification Language ASL [WKC01] oder der „Action Language“-Teilmenge von SDL [Leb04] auf die Instanzen der Metatypen demonstriert (siehe z.B. [OMG03, S.B-1ff.]). Der

Vorteil dieses Ansatzes der OMG liegt hierbei darin, dass alle in der Praxis eingesetzten Aktionssprachen weiterbenutzt werden können, solange eine solche Abbildung auf das Metamodell der UML Action Semantics möglich ist. Insbesondere ist auch die Abbildung einer Programmiersprache (oder zumindest deren Teilmenge [MTA99]) auf das Metamodell der Action Semantics denkbar [SPH01].

In dieser Arbeit wollen wir eine eigene Aktionssprache einführen, um die im weiteren Verlauf der Arbeit vorgestellten konkreten Aktivitäten präzise aber dennoch abstrakt genug beschreiben zu können, ohne sich in Implementierungsdetails zu verlieren. Wir wollen diese Sprache *AL++* (*Action Language ++*) nennen, da sie sich nahtlos in unsere verbesserte (+) Multiebenenmodellierung (+) eingliedert. Obwohl man eine solche Sprache direkt zum Zweck der Spezifikation und damit zur Generierung der Verhaltensaspekte der Automatisierungswerkzeuge einsetzen könnte, wurde auf die Erstellung eines solchen Generators (oder AL++-Compilers) verzichtet, da dies den Rahmen der Arbeit gesprengt hätte.

Aus demselben Grund wird diese Sprache auch nicht vollständig definiert und formalisiert, sondern der Einfachheit und Verständlichkeit halber eine um spezifische und implementierungsunabhängige OO-Konzepte erweiterte Teilmenge der Sprache *Java* gewählt (für eine Einführung in diese Sprache siehe z.B. [Poe00] oder [MSS96]). Die Abbildung der wichtigsten Aspekte dieser Aktionssprache auf ausführbaren Code werden wir in Abschnitt 5.2 auf Seite 96 vorstellen.

Als Teilmenge wählen wir all diejenigen Konstrukte, die eine algorithmische Verhaltensbeschreibung erlauben, also Kontrollkonstrukte, Operationsaufrufe, Berechnungen, usw. Die existierenden Java-Konstrukte werden – wenn nicht anders angegeben – in ihrer üblichen Java-Semantik interpretiert. Die neuen Konstrukte werden wir in den folgenden Unterabschnitten erläutern.

Als laufendes Beispiel nehmen wir wieder einen Ausschnitt aus dem M_1-Modell des Grafikbeispiels. Zur einfacheren Lesbarkeit ist dieser Ausschnitt in Abb. 4-3 wiederholt. Desweiteren führen wir Rollennamen für die beschriebenen Assoziationen ein, die einen eindeutigen Zugriff auf die Links von Seiten der Aktionssprache erlauben.

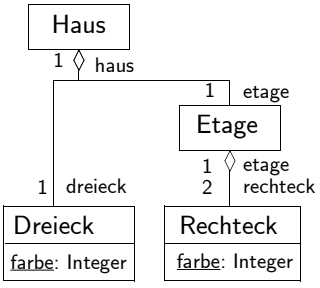


Abbildung 4-3. Ausschnitt aus dem Beispielmodell

Handhabung von Objekten

Zur Erzeugung von Objekten wird das Schlüsselwort `new` verwendet. Ebenso werden *Objektreferenzen*, also Verweise auf die Objektinstanzen, wie in Java eingesetzt (siehe z.B. [Poe00, S.23f.]). Damit lässt sich dann z.B. eine neue Instanz des Objekttyps Dreieck, auf die mit der Referenz `meinDreieck` verwiesen wird, erzeugen durch:

```
Dreieck meinDreieck = new Dreieck();
```

In Übereinstimmung mit der Java-Syntax bietet unsere Sprache – anders als z.B. die oben erwähnte ASL – keine implizite Variablendeklaration an, weshalb der Typ der Objektreferenz (hier `Dreieck`) angegeben werden muss.

Zum Entfernen eines Objekts muss man lediglich dafür sorgen, dass keine Referenzen auf dieses Objekt verweisen und dass das Objekt in keiner Attributbelegung und keiner Assoziation (Link) vorkommt. Damit übernehmen wir hier den *Garbage-Collector*-Mechanismus von Java (siehe z.B. [MSS96, S.64]). In obigem Beispiel würde es prinzipiell genügen `meinDreieck = null` zu setzen (vorausgesetzt die Aggregations-Links wären noch nicht belegt).

Handhabung von Attributen

Die Belegung eines Attributs (genauer einer Attributinstanz) mit einem Wert erfolgt mit der üblichen Notation von Java, also z.B. durch

```
meinDreieck.farbe = 0;
```

oder direkt durch

```
farbe = 0;
```

falls es sich um eine Aktion innerhalb des Objekts handelt, das dieses Attribut besitzt.

Analog erfolgt das Auslesen eines Attributwerts z.B. mit

```
Integer meineFarbe = meinDreieck.farbe;
```

Handhabung von Links

Der Zugriff auf Links erfolgt stets über die Rollennamen der Assoziation. Damit ergibt sich eine zum Attributzugriff analoge Notation, setzt aber voraus, dass die entsprechenden Rollennamen definiert wurden.

In Abschnitt 3.1.1 auf Seite 31 hatten wir bereits Rollen als Abbildung zwischen der Extension des Typs auf der Ursprungsseite und der Potenzmenge der Extension des Typs auf der Zielseite der Assoziation interpretiert (siehe Gl. 3-4 auf Seite 33).

Damit wäre das Ergebnis eines Zugriff auf ein Link-Ende über die Rolle stets eine Menge.

Zur weiteren Reduktion des Spezifikationsaufwandes wollen wir das Ergebnis eines Zugriffs in Abhängigkeit von der Multiplizität des Assoziationsendes einschränken. Konkret bedeutet dies, dass bei einer Multiplizität von „0..1“ oder „1“ immer eine Instanz des Typs zurückgeliefert wird und man nur bei Multiplizitäten größer „1“ eine Menge erhält.

Im obigen Beispiel hieße dies, dass folgende Zugriffe möglich (und gültig wären):

```
Dreieck meinDreieck = meinHaus.dreieck;
```

und

```
Set meineRechtecke = meineEtage.rechteck;
```

Auf die zurückgelieferte Menge kann man mit den entsprechenden Mengenoperationen wie sie für `java.util.Set` definiert werden zugreifen.

Als weitere Vereinfachung des Zugriffs auf Assoziationsenden, die mehr als eine Instanz beinhalten, führen wir ein `foreach`-Kontrollkonstrukt ein. Dieses erlaubt das Iterieren über die Elemente der angegebenen Menge. Die Syntax ist wie folgt:

```
foreach(<objektReferenz> in <mengenReferenz>)
    <anweisungsBlock>
```

Die `<objektReferenz>` bezeichnet dabei eine Laufvariable, die innerhalb des Anweisungsblocks verwendet wird. Bei mehr als einer Anweisung wird dieser Block – wie von Java bekannt – in geschweifte Klammern eingeschlossen. Die `<mengenReferenz>` schließlich gibt den `Set` an, über dessen Elemente iteriert werden soll.

Um eine konkurrente Modifikation der Menge, über die iteriert wird, zu vermeiden, fordern wir – wie auch in der Aktionssprache ASL –, dass die Menge während der Iteration nicht geändert werden darf (siehe [WKC01, S.17]). Für die typischen Operationen bei einer Modelltransformation stellt dies allerdings keine bedeutende Einschränkung dar, da Änderungen der an einer Assoziation beteiligten Instanzen (mit Hilfe der weiter unten folgenden Aktionen) weiterhin zulässig sind. Anders formuliert liefert der Zugriff auf ein Link-Ende (mit Multiplizität größer „1“) immer eine *Kopie* der Menge der Instanzen zurück.

In unserem Beispiel ließe sich mit diesem Konstrukt sehr einfach eine Farbänderung aller von einer Etage `meineEtage` aggregierten Rechtecke beschreiben durch:

```
Set meineRechtecke = meineEtage.rechteck;
Rechteck aktuellesRechteck;
foreach(aktuellesRechteck in meineRechtecke)
    aktuellesRechteck.farbe = 1;
```

Wie in Java kann man das auch verkürzt schreiben als:

```
foreach(Rechteck aktuellesRechteck in meineEtage.rechteck)
    aktuellesRechteck.farbe = 1;
```

Das Erzeugen neuer Links erfolgt ebenfalls über die Nutzung der Rollennamen. So erzeugt der Ausdruck

```
<objektReferenz1>.<rollenBezeichner> += <objektReferenz2>;
```

einen neuen Link, der das Objekt, gekennzeichnet durch `<objektReferenz1>`, mit dem Objekt, gekennzeichnet durch `<objektReferenz2>`, verbindet. Wie schon bei den Attributen kann auch hier die `<objektReferenz1>` entfallen, wenn es sich um eine aus dem Kontext der aktuellen Instanz (`this`) bekannte Rolle handelt.

Bei Multiplizitäten größer eins impliziert die Interpretation von Rollen als Abbildung auf Mengen, dass niemals mehr als *ein* identisches Objekt in einer solchen Menge auftauchen kann. Das heißt, wird versucht einen bereits existierenden Link erneut hinzuzufügen, dann ändert sich durch diese Aktion nichts an den Mengen der Links.

Bei Multiplizitäten bis einschließlich eins (die Erzeugung der Links erfolgt in diesem Fall mit „=“ an Stelle von „+=“) ist eine solche Semantik nicht aus der Definition abzuleiten. Wir legen daher fest, dass wenn bereits ein Objekt am Assoziationsende einer zu-1-Assoziation existiert, dieses durch das Hinzufügen eines neuen Links ersetzt wird (siehe dazu auch [OMG03, S.B-15]).

Zuletzt benötigt man noch ein Konstrukt zum Entfernen oder Löschen von Links. Hier verwenden wir die Notation

```
<objektReferenz1>.<rollenBezeichner> -= <objektReferenz2>;
```

zum Löschen eines einzelnen Links. Mit der Angabe von

```
<objektReferenz>.<rollenBezeichner> -= all;
```

werden alle Links, auf die mit `<rollenBezeichner>` zugegriffen werden kann, gelöscht.

Alle diese Aktionen sind übrigens robust gegenüber einer `null`-Referenz, also einer Referenz, die auf kein Objekt verweist. So zeigt das Hinzufügen oder Entfernen eines Links mit einem `null`-Handle keinen Effekt.

In Abschnitt 3.1.1 auf Seite 31 hatten wir bereits darauf hingewiesen, dass man Assoziationen in gerichtete oder bidirektionale unterscheiden kann. Für bidirektionale Relationen wollen wir die Bedeutung der obigen Konstrukte ergänzen: Durch das Hinzufügen eines Objekts zu der Menge auf der Zielseite einer bidirektionalen Assoziation wird automatisch auch die Instanz, aus dessen Kontext diese Aktion

aufgerufen wurde, in die Menge der Objekte auf der Ursprungsseite der Assoziation eingetragen („referentielle Integrität“).

Im Grafikbeispiel bedeutet dies, dass sowohl

```
meinHaus.dreieck = meinDreieck;
```

als auch

```
meinDreieck.haus = meinHaus;
```

dieselben Mengen erzeugt, nämlich

`meinHaus.dreieck` = {`meinDreieck`} und `meinDreieck.haus` = {`meinHaus`}.

Multiebenenbeschreibung

Die oben beschriebenen Operatoren sind in gleicher oder ähnlicher Form auch in den existierenden Aktionssprachen enthalten. Mit der Einführung der verbesserten Multiebenenmodellierung reichen diese Konstrukte aber nicht mehr aus. In den folgenden Abschnitten werden wir daher unsere diesbezüglichen Erweiterungen der Aktionssprache AL++ vorstellen.

Als Erstes legen wir dazu fest, dass alle Objekttypen ein vordefiniertes Attribut name besitzen, das bei der Instanziierung des Objekttyps belegt werden muss (dabei handelt es sich um die bereits in Abschnitt 4.1 auf Seite 55 erwähnte Instanziierungsinformation). Daher erfolgt die Instanziierung von Objekttypen einer Potenz größer zwei über

<M_n-ObjektReferenz> = new <M_{n+1}-ObjektReferenz>(<M_n-ObjektName>)

Für das Grafikbeispiel würde man so zum Beispiel den Typ Dreieck der Ebene M_1 durch

```
SimplerGrafikElementTyp TypDreieck =
 new SimplerGrafikElementTyp("Dreieck");
```

erzeugen, wobei in der Variable `TypDreieck` eine Referenz auf den erzeugten Typ gehalten wird. Dies ist analog zu den Objektreferenzen in Java.

Multiebenenbeschreibung von Relationen

Neben Objekttypen lassen sich bei der verbesserten Multiebenenmodellierung auch andere Modellierungselemente über mehrere Ebenen hinweg instanziieren. Dies gilt insbesondere für Relationen, die eine Potenz von mindestens zwei besitzen. Zur operationalen Bearbeitung solcher „höherpotenten“ Relationen müssen wir den oben vorgestellten Mechanismus für den Umgang mit Links erweitern.

Für eine vollständige Instanziierung einer Relation mit einer Potenz größer eins, benötigt man zusätzliche Informationen. Dazu gehört der Name der Relationsinstanz, die Multiplizitäten der Enden und die Rollennamen. Zur Angabe dieser Informationen werden diese als zusätzliche Parameter bei der Erzeugung einer Relation der Ebene M_n in der folgenden Form angegeben:

```
<M_n-ObjektReferenzUrsprung>.<M_n/n+1-RollenBezeichner> +=
 <M_n-ObjektReferenzZiel>, (<M_n-1/n-RelationsName>,
 <M_n-1/n-MultiplizitaetUrsprung>, <M_n-1/n-MultiplizitaetZiel>,
 <M_n-1/n-RollenNameUrsprung>, <M_n-1/n-RollenNameZiel>);
```

Bei `<...Name>` handelt es sich dabei stets um einen String, welcher in „"“ eingeschlossen wird, wohingegen es sich bei `<...Bezeichner>` um reinen Text zur eindeutigen Referenzierung eines Modellelements handelt. Durch $M_{v/d}$ wird jeweils die Verwendungsebene (M_v) und Definitionsebene (M_d) eines Elements angegeben, womit dessen eindeutige Einordnung in die Metahierarchie möglich ist.

Die Angabe der Rollennamen und Relationsnamen ist verbindlich, wenn man später auf die Links der so erzeugten Relation zugreifen will. Die Angabe inkonsistenter oder konfliktionärer Informationen (so z.B. die Änderung der Multiplizität auf „1“, wenn schon mehr als ein M_{n-1}-Objekt am Link-Ende existiert) ist nicht erlaubt.

Für die Angabe von Multiplizitäten sehen wir den Datentyp `Multiplicity` vor, dessen Literale die von der UML bekannten Multiplizitätsangaben sind (diese hatten wir bereits mehrfach verwendet). Es lässt sich allerdings auch einzeln auf die die untere Grenze mittels `lower` bzw. auf die obere Grenze mittels `upper` zugreifen.

Da mit der Einführung der Multiebenenbeschreibung das übliche Namensschema von Java (kleiner Anfangsbuchstabe für Instanzen, großer für Typen) nicht mehr ausreicht, können wir die Identifier in der Sprache durch ein entsprechendes Subskript ergänzen, welches die Modellebene kennzeichnet. Dieses Subskript hat rein erklärenden Charakter und ist nicht Teil des Namens (es kann also auch weggelassen werden).

Abb. 4-4 zeigt ein mögliches Modell (untere Hälfte der Abbildung), das durch die Instanziierung eines Metamodells (obere Hälfte der Abbildung) entstanden ist. Diese Instanziierung wird durch das folgende Code-Fragment beschrieben (zur besseren Verständlichkeit ist hier die Ebene, auf der sich ein Modellelement befindet, angegeben):

```
ElementTyp_2 TypFirma_1 = new ElementTyp_2("Firma");
ElementTyp_2 TypPerson_1 = new ElementTyp_2("Person");
TypFirma_1.ziel += TypPerson_1, ("stelltAn", 1, 0..*,
 "anstellendeFirma", "angestellter");
```

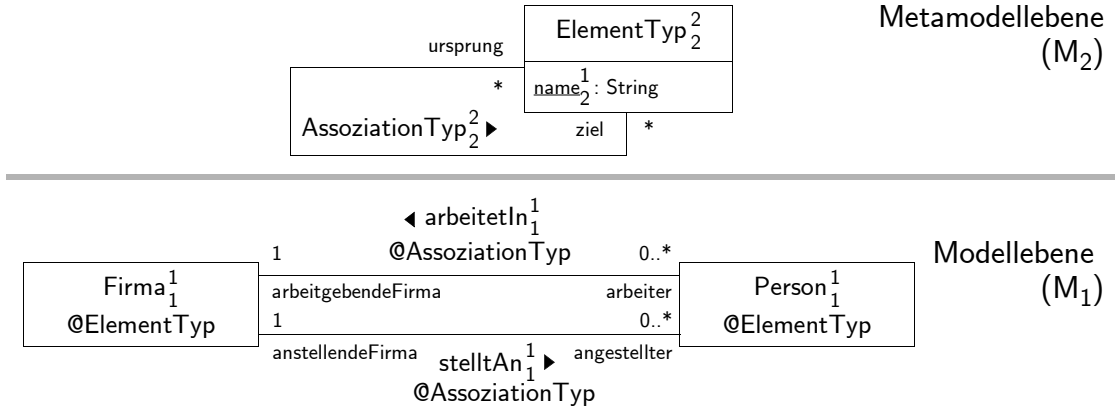


Abbildung 4-4. Beispiel zur Instanziierung von Relationen (mit Metamodell)

```
TypPerson1.ziel += TypFirma1, ("arbeitetIn", 0..*, 1,
 "arbeiter", "arbeitgebendeFirma");
```

Nach einer solchen Instanziierung würde man dann mit den Mitteln der Aktionssprache auf die Links der Relation arbeitetIn und stelltAn zugreifen. Da die Rollennamen durch eine solche dynamische Instanziierung allerdings erst zur „Laufzeit“ bekannt sind, führen wir den „generischen“ Operator „#“ für den Zugriff auf Rollen und Attribute ein. Sei `peter` eine Objektreferenz auf eine Instanz von `Person`, dann kann man mit

```
String rollenName = "arbeitgebendeFirma";
Firma1 petersFirma0 = peter0.#rollenName;
```

auf das Link-Ende zugreifen.

Neben der Erzeugung neuer Relationen ist natürlich auch ein Zugriff auf die Eigenschaften existierender Relationen notwendig, was durch

```
(<Mn-1/n-RelationsName>, <Mn-1/n-MultiplizitaetUrsprung>,
 <Mn-1/n-MultiplizitaetZiel>, <Mn-1/n-RollenNameUrsprung>,
 <Mn-ObjektReferenzZiel>) =
 (<Mn-ObjektReferenzUrsprung>.<Mn/n+1-RelationsBezeichner>,
 <Mn-1/n-RollenNameZiel>);
```

ermöglicht wird. Die Angabe von `<Mn-1/n-RollenNameZiel>` erlaubt die eindeutige Auswahl der gewünschten Relation aus der Menge aller instanziierten Relationen. Durch Angabe von `null` für einen der Rückgabeparameter kann man auf die Rückgabe des entsprechenden Wertes verzichten.

Mit dem folgenden Code-Fragment erhält man dann also z.B. die Multiplizität der stelltAn-Relation auf der Seite von Person zurück:

```
Multiplicity multiplizitaet;
(null, null, multiplizitaet, null, null) = (TypFirma1.AssoziationTyp,
```

```
 "angestellter");
```

Kennt man den Rollennamen am Ziel des Relationstyps noch nicht, so kann man alle Rollennamen eines gegebenen Relationstyps ausgehend von einem Ursprungsobjekttyp wie folgt bestimmen:

```
<mengenReferenz> = <M_n-ObjektReferenzUrsprung>.
 <M_n/n+1-RelationsBezeichner>;
```

Die zurückgelieferte Menge beinhaltet dann alle Rollennamen auf der Seite der über die $M_{n/n+1}$-Relation mit dem Ursprungsobjekttyp verbundenen Instanzen. So würde im Beispiel der Aufruf von `TypFirma.AssoziationTyp` die Menge mit den Elementen `"angestellter"` und `"arbeiter"` liefern.

Diese Menge von Rollennamen lässt sich bei Bedarf weiter präzisieren, indem man nicht den Typ der Relation angibt, sondern die Richtung, in welcher die möglichen Relationsinstanzen betrachtet werden sollen:

```
<mengenReferenz> = <M_n-ObjektReferenzUrsprung>.
 <M_n/n+1-RollenBezeichner>;
```

Im Beispiel erhielte man so für den Aufruf von `TypFirma.ziel` lediglich eine Menge mit dem Eintrag `"angestellter"` zurück.

Multiebenenbeschreibung von Attributen

So wie die obigen Relationstypen lassen sich auch Attributtypen explizit instanziieren (dies ist eine Alternative zu der automatischen Instanziierung, welche bei der tiefen Instanziierung vorgeschlagen wurde, siehe Abschnitt 4.1 auf Seite 55).

Für einen Attributtyp, der innerhalb eines Objekttyps der Ebene M_{n+1} definiert wurde, lautet die allgemeine Syntax für die Erzeugung neuer M_n-Attribute

```
<M_n-ObjektReferenz>.<M_n/n+1-AttributTypBezeichner> +=
 (<M_n-1/n-AttributName>, <M_n-DatenTypReferenz>);
```

Mit Hilfe der folgenden Zeilen würde man zum Beispiel den Objekttyp 2DPunkt mit seinen Attributen x und y aus Abb. 4-2 auf Seite 60 erzeugen.

```
PunktTyp Typ2DPunkt = new PunktTyp("2DPunkt");
Datentyp TypInteger = /* ... */;
Typ2DPunkt.AttributTyp += ("x", TypInteger);
Typ2DPunkt.AttributTyp += ("y", TypInteger);
```

Die Menge aller Namen der Instanzen eines gegebenen Attributtyps innerhalb des Kontext-Objekttyps erhält man – in einer analogen Syntax zu obigem Relationszugriff – mit

```
<mengenReferenz> = <M_n-ObjektReferenz>.
 <M_n/n+1-AttributTypBezeichner>;
```

Schließlich lässt sich der Datentyp eines bestimmten Attributs (analog zu der Bestimmung von Rollen und Multiplizitäten bei Relationen) wie folgt ermitteln:

```
<M_n-DatentypReferenz> =
 (<M_n-ObjektReferenz>.<M_n/n+1-AttributTypBezeichner>,
 <M_n-1/n-AttributName>);
```

Zuletzt definieren wir auch für Attribute den „`#`“-Operator für den Laufzeitzugriff auf dynamisch erzeugte Attribute.

Sonstige Konstrukte der Multiebenenbeschreibung

Um die Multiebenenmodellierung voll nutzen zu können, bedarf es schließlich noch der Unterstützung der „Navigation“ zwischen den Metaebenen in AL++. Der Operator, der dies erlaubt ist `type`, der ähnlich zu der `getClass()`-Methode in Java, den Typ der Instanz zurückliefert. Allgemein gilt also:

```
<M_n+1-ObjektReferenz> = <M_n-ObjektReferenz>.type;
```

Hierbei liefert der `type`-Operator exakt *den* Typ zurück, der instanziiert wurde (und nicht einen eventuellen Supertyp). Angenommen `T` sein ein Typ, `t` eine Instanz dieses Typs und `ST` der Supertyp von `T`, dann würde der Vergleich `t.type == ST` ein `false` als Ergebnis liefern. Will man prüfen, ob eine Instanz von einem entsprechenden Supertyp ist, so muss man den von Java bekannten `instanceof`-Operator einsetzen.

Wie wir in Abschnitt 4.1 auf Seite 55 detailliert erläuterten, erübrigt sich bei Einsatz der verbesserten Multiebenenmodellierung die Definition eines Supertyps zur Einordnung von Modellelementen mit gemeinsamen Eigenschaften. Dies stellt nun leider wieder ein Problem für unsere Aktionssprache dar, da wir an vielen Stellen – wie auch in Java üblich – eine generische Verhaltensbeschreibung vornehmen möchten. So braucht man den konkreten Typ eines simplen Grafikelements aus Abb. 4-1 auf Seite 59 (also z.B. Dreieck) nicht zu spezifizieren, wenn man nur dessen Farbe erfahren möchte, da alle Instanzen von SimplerGrafikElementTyp ein entsprechendes Attribut besitzen. Um solch eine generische Beschreibung zu ermöglichen, führen wir daher den speziellen Operator `anyinstance` ein, der uns einen „gedachten“ Supertyp (als Instanz des spezifizierten Metatyps) liefert:

```
<M_n+2-ObjektReferenz>.anyinstance <M_n-ObjektReferenz>
```

Folgendes Code-Fragment würde dann das obige Beispiel realisieren:

```
SimplerGrafikElementTyp_2.anyinstance einGrafikElement_0 = /* ... */;
```

```
Integer eineFarbe = einGrafikElement0.farbe;
```

Bei der Vorstellung der Operationen für Relationen und Attribute hatten wir bereits den „#“-Operator zur Realisierung eines generischen Zugriffs eingeführt. An dieser Stelle wollen wir dessen Einsatzgebiet auch auf Objekttypen erweitern. So kann man z.B. mit

```
String typName = "Person";
ElementTyp2.anyinstance meinePerson0 = new #typName();
```

zur Laufzeit eine Instanz des Typs Person erzeugen. Desweiteren kann man mit Hilfe des „#“-Operators die Typen der Objektreferenzen auch generisch spezifizieren. Die letzte Zeile des obigen Beispiels kann man daher auch schreiben als:

```
#typName meinePerson0 = new #typName();
```

Den Einsatz der in diesem Abschnitt eingeführten Multiebenenoperatoren werden wir Abschnitt 5.2.4 auf Seite 109 an einem größeren Beispiel veranschaulichen. In Anhang A findet der Leser eine kompakte Zusammenfassung aller Sprachkonstrukte der AL++.

4.2.3 Modelltransformationen auf den Ebenen M_0 bis M_3

So wie zu Beginn von Kapitel 3 Beispiele für OO-Modelle verschiedener Metaebenen vorgestellt wurden, wollen wir nun auch Transformationen auf den verschiedenen Ebenen an Hand des Grafikbeispiels illustrieren.

Transformation auf Ebene M_0

Betrachtet man die Menge von Laufzeitinstanzen (inklusive deren Attributbelegungen und Links) als Zustand eines ausgeführten Software-Systems, dann handelt es sich bei einer Transformation auf dieser Ebene um die eigentliche Programmausführung, oder zumindest einen Schritt in dieser Ausführung.

Bei unserem Grafikprogramm wäre dies z.B. das Erzeugen einer neuen „Hausinstanz“ zur Laufzeit. Beschrieben wird diese Erzeugung – wie wir es z.B. von einer UML-Modellierung von Software-Systemen her kennen – auf der M_1-Ebene. Diese Ebene nutzten wir bereits zur Veranschaulichung der Konstrukte unserer Aktionssprache.

An dieser Stelle wollen wir ein komplexeres Beispiel präsentieren, das es erlaubt eine Instanz eines Haus-Typs um einen bestimmten Offset zu verschieben. In Abb. 4-5 ist das um die benötigten Operationen erweiterte M_1-Modell gezeigt.

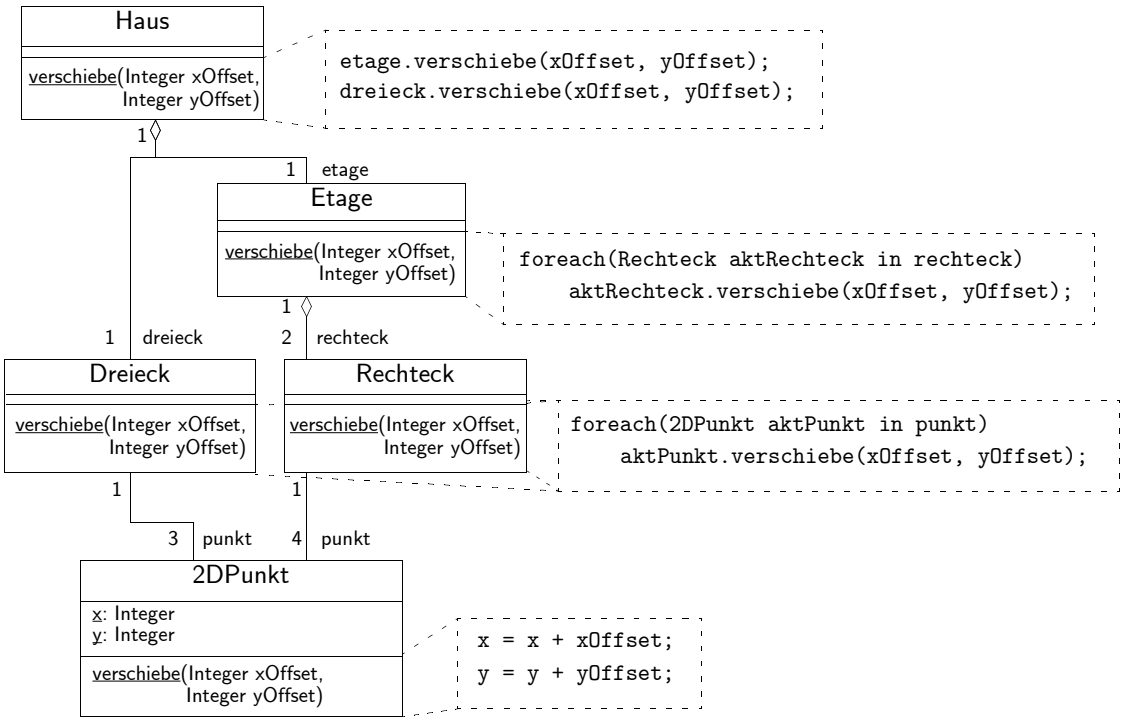


Abbildung 4-5. Beispiel einer M_1-Operation (Operationen für Dreieck und Rechteck sind gleich)

Durch den Aufruf der Operation `verschiebe()` einer Instanz des Objekttyps Haus werden alle Punkte so um den angegeben Offset verschoben, dass die gesamte Hausinstanz verschoben wird.

Transformation auf Ebene M_1

Die Elemente der Ebene M_1 sind die eigentlichen Entwicklungsartefakte, die es bei einer Software-Entwicklungsaktivität zu bearbeiten gilt (siehe Abschnitt 2.2.2 auf Seite 19). Dies bedeutet auch, dass die durch eine solche Aktivität bewirkte Modelltransformation auf der Ebene M_2, also auf der Metamodellebene, beschrieben werden muss (siehe dazu auch [MSU02]).

In Abb. 4-6 ist das Schema einer solche M_1-Modelltransformation gezeigt. Es handelt sich hierbei um eine Erweiterung von Abb. 2-7 auf Seite 20.

Hier zeigt sich die Mächtigkeit der Metamodellierung. Denn genauso wie wir die Instanzen des M_1-Modells durch unsere Aktionssprache AL++ bearbeiten und somit das Verhalten des Laufzeitsystems beschreiben konnten, können wir jetzt mit derselben Sprache eine Änderung der Instanziierung eines M_2-Modells und damit eine Entwicklungsaktivität spezifizieren. Durch unseren operationalen Ansatz kön-

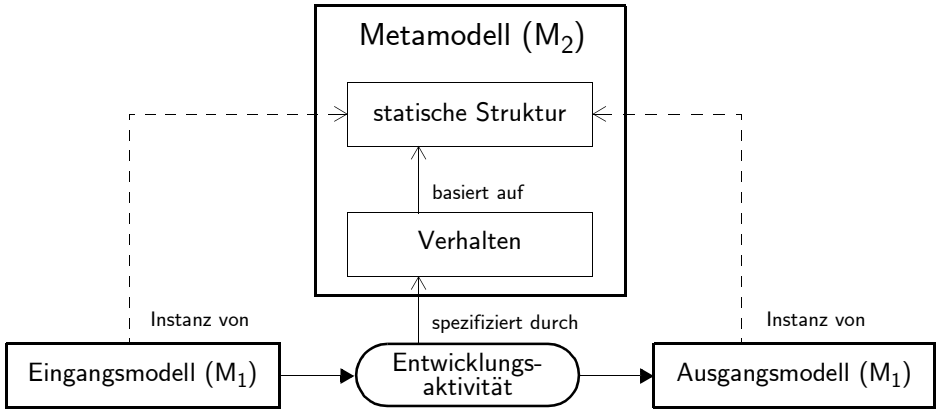


Abbildung 4-6. Rolle von Metamodell und Modell bei einer Modelltransformation

nen wir also jede Modelltransformation genauso „programmieren“ wie wir eine „normale“ Software-Applikation realisieren würden.

Man könnte sich in unserem laufenden Beispiel z.B. vorstellen, dass ein Modellierer den Typ Kreis gegen den allgemeineren Typ Ellipse austauschen möchte, da dafür ein Algorithmus zur Darstellung auf dem Bildschirm existiert.

In AL++ lässt sich dies als Operation `tauscheKreis()` des Metatyps KomplexerGrafikElementTyp wie folgt implementieren (wir gehen hierbei davon aus, dass ElementAggregationTyp die Rollennamen ganzes und teil besitzt und der Rollennamen am Ziel des PunktAssoziationsTyps punkttyp ist):

```
1   /* KomplexerGrafikElementTyp: */
2   tauscheKreis() {
3       GrafikElementTyp2 MeinTeil1;
4       foreach(MeinTeil1 in teil) {
5           if(MeinTeil1.type == KomplexerGrafikElementTyp2) {
6               ((KomplexerGrafikElementTyp2)MeinTeil1).tauscheKreis();
7           } else {
8               SimplerGrafikElementTyp2 MeinSimplerTeil1 =
9                 (SimplerGrafikElementTyp2)MeinTeil1;
10              if(MeinSimplerTeil1.name == "Kreis") {
11                  PunktTyp2 MeinPunktTyp1 = MeinSimplerTeil1.punktTyp;
12                  MeinSimplerTeil1.punktTyp -= MeinPunktTyp1;
13
14                  SimplerGrafikElementTyp2 MeineEllipse1 =
15                    new SimplerGrafikElementTyp2("Ellipse");
16                  MeineEllipse1.punkttyp += MeinPunktTyp1,
17                    ("Assoziation", 1, 3, null, null);
18
19                  foreach(KomplexerGrafikElementTyp2 MeinGanzes1
20                    in MeinSimplerTeil1.ganzes) {
```

```
21                          MeinGanzes1.teil -= MeinSimplerTeil1;
22                          MeinGanzes1.teil += MeineEllipse1,
23                            ("Aggregation", 1, 1, null, null);
24                      }
25                  }
26              }
27          }
28      }
```

Ausgehend von einer Instanz des KomplexenGrafikElementTyps, aus deren Kontext die Operation aufgerufen wird, laufen wir zunächst über alle aggregierten Instanzen (Rollenname teil). Da diese entweder komplexe oder simple Grafikelemente sein können, wählen wir als Typ der Laufvariable den Supertyp GrafikElementTyp. Falls es sich um eine Instanz vom Typ KomplexerGrafikElementTyp handeln sollte (Zeile 5), dann rufen wir wiederum die `tauscheKreis()`-Operation auf. Allerdings müssen wir zunächst eine Re-Typisierung der Objektreferenz auf den Typ KomplexerGrafikElementTyp vornehmen. Dazu verwenden wir den von Java bekannten „Cast“-Operator (siehe z. B. [Eck02, Kapitel 3]).

Ansonsten kann es sich dann nur noch um einen simplen Grafikelementtyp handeln und wir führen wieder eine entsprechende Re-Typisierung durch. Danach prüfen wir, ob es sich bei der Instanz um eine mit dem Namen „Kreis“ handelt und damit dem Modellelement entspricht, das wir ersetzen wollen.

Bevor wir den eigentlichen Austausch vornehmen, besorgen wir uns eine Referenz auf die Instanz von PunktTyp, die für alle unsere Grafikelemente benötigt wird. Zusätzlich entfernen wir die Assoziation dieses Punkttyps zu Kreis (Zeile 12). Jetzt können wir unser neues Grafikelement erzeugen. Dazu instanziieren wir SimplerGrafikElementTyp (Zeile 15) und vergeben in der folgenden Zeile den entsprechenden Namen. Nun müssen wir noch die Assoziation zu den Punkten der Ellipse anlegen. Dies ist eine Assoziation vom Typ PunktAssoziationTyp (was durch den Rollennamen punkttyp eindeutig beschrieben ist). Da wir anders als bei einem Kreis, für eine Ellipse drei Punkte benötigen (den Mittelpunkt und die zwei Brennpunkte), handelt es sich bei der Multiplizität des Assoziationsendes um „3“.

Zuletzt muss dieser neue Typ noch in die Modellstruktur eingebunden werden. Dazu werden zunächst die eventuellen Assoziationen zu anderen komplexen Grafikelementen entfernt (Zeile 21) und danach die neue Assoziation eingefügt. Damit genügt es, diese Operation für eine beliebige Instanz von KomplexerGrafikElementTyp aufzurufen, die einen Kreis aggregiert, um alle Assoziationen zu diesem Kreis ent-

sprechend auf Assoziationen zur Ellipse zu ersetzen. In Abb. 4-7 ist der Verlauf dieser Operation an dem relevanten Ausschnitt des Modells gezeigt.

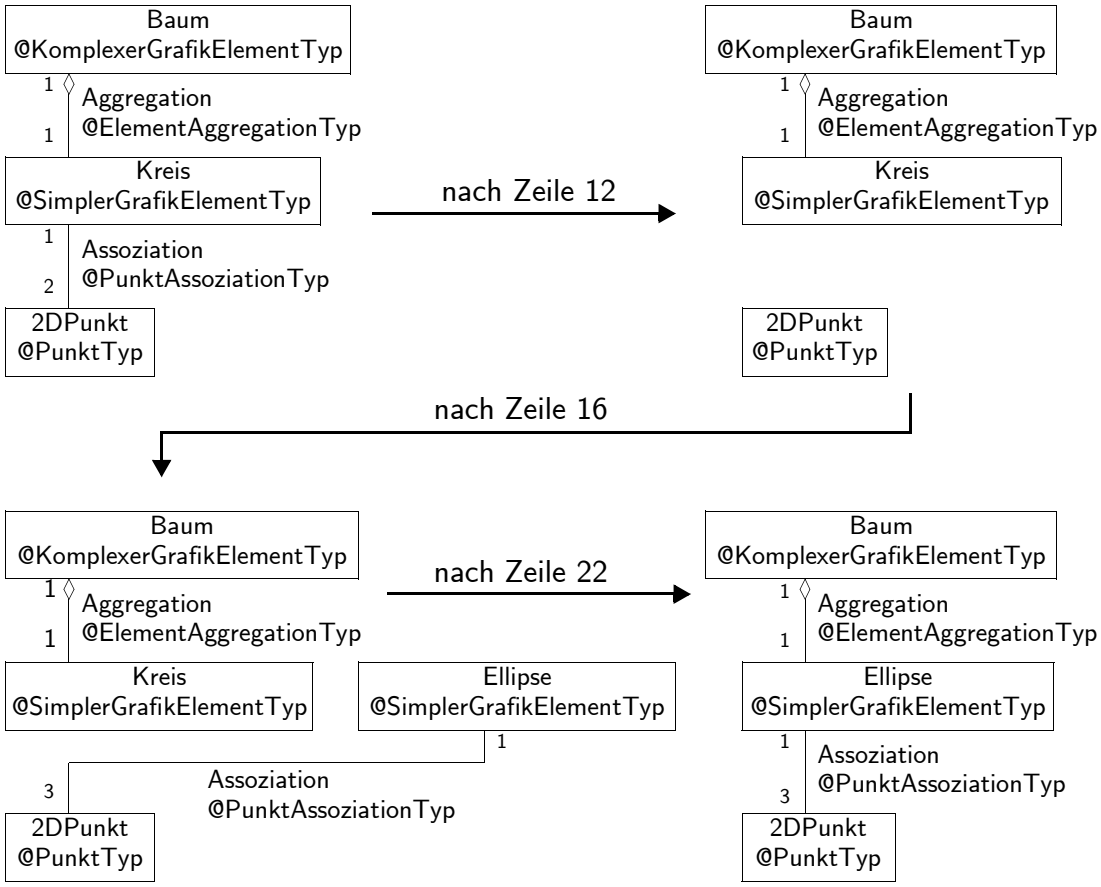


Abbildung 4-7. Modelltransformation im Grafikbeispiel

Anders als eine Transformation auf Ebene M_1 können die Transformationen auf einer höheren Metaebene weitreichende Konsequenzen für die ausgehend von diesem Modell erzeugten Modellinstanzen haben. So muss allgemein nach einer Änderung der M_n-Elemente sichergestellt sein, dass sich alle Modelle der Ebenen kleiner n nach wie vor beschreiben lassen. Insbesondere wenn man die Modelle der niedrigeren Modellebenen beibehalten möchte, ist dafür zu sorgen, dass diese Modelle an das neue Metamodell angepasst werden. Für unser oben zitiertes Beispiel bedeutet dies, dass alle Instanzen vom Typ Kreis gegen die entsprechenden Instanzen vom Typ Ellipse „ausgetauscht“ werden müssten. Insofern ähnelt dieses Problem dem Problem der *Schema-Evolution*, das man aus Datenbanken kennt (siehe dazu z.B. [DLR03], [Ter01] und [FGM01]).

Transformation auf Ebene M_2 und M_3

Auf der Ebene M_2 findet eine Transformation der Metatypen (oder Meta-Metainstanzen) statt. Prinzipiell lassen sich auch auf dieser Ebene ähnliche Operationen

wie die bereits vorgestellten durchführen, mit der Konsequenz, dass eventuell alle darunterliegenden Modelle angepasst werden müssen.

Änderungen auf der Ebene M_3 betreffen (zumindest in unserem Beispiel) die grundlegenden Begriffe der verwendeten OO-Sprache und zeigen deshalb die weitreichendsten Konsequenzen. Eine Anpassung aller abgeleiteten Modelle stellt damit einen sehr großen Aufwand dar, der nicht unbedingt effizient automatisiert werden kann, da auch das Erstellen der Werkzeuge nicht einfach ist. Besonders bei Semantikänderungen kann ein solches Unterfangen sehr aufwändig werden. Holz deutet in [Hol03] die prinzipiellen Probleme an, die sich bei einer solchen Sprachübersetzung auftun. Man sollte daher tunlichst die Vorteile einer Abbildung auf dieser Ebene gegen die Nachteile, diese Abbildung nicht vorzunehmen, abwägen.

Zusammenfassung

Aufbauend auf einer verbesserten Multiebenenmodellierung (als Lösung des „flachen“ Instanziierungsproblems) wurden die vereinfachte und systematische Modellierung des „Grafik“-Beispiels des vorhergehenden Kapitels präsentiert.

In der zweiten Hälfte des Kapitels wurde die statische Metamodellierung um die Aspekte der Modelltransformation erweitert. Erreicht wurde dies durch Ausnutzen der OO-Konzepte zur Verhaltensbeschreibung. Speziell wurde eine algorithmische Beschreibung mit Hilfe der Aktionssprache AL++ eingeführt, welche eine implementierungsunabhängige und operationale Transformation von Modellen erlaubt.

Damit stellt dieses Kapitel die Grundlage für das nun folgende Kapitel dar, in welchem wir die eingeführten Konzepte für einen konkreten Automatisierungsansatz zur Anwendung bringen werden.

5 Eine systematische Technik zur Automatisierung

Die Rechenautomaten haben etwas von den Zauberern im Märchen. Sie geben einem wohl, was man sich wünscht, doch sagen sie einem nicht, was man sich wünschen soll.

— Norbert Wiener (amerik. Mathematiker, Begründer der Kybernetik, 1894–1964)

In den vorangegangenen Kapiteln hatten wir bereits die Mächtigkeit des Metamodellansatzes für eine Automatisierung von Entwicklungsaktivitäten aufgezeigt. Insbesondere gelang es uns durch die Einführung einer Metaebene die Entwicklungsaktivitäten als Transformation von Produktmodellinstanzen zu betrachten, weshalb die Beschreibung solcher Modelltransformationen durch die OO-Konstrukte zur operationalen Verhaltensbeschreibung möglich wurde.

Bisher wurden diese Transformationen allerdings nur für die abstrakten Notationselemente eingeführt. In diesem Kapitel wollen wir nun zusätzlich aufzeigen, wie man systematisch von einer konkreten Repräsentation zu einer abstrakten Darstellung gelangt und umgekehrt. Dazu wird ein Metamodell eingeführt, das die Klassifikation der konkreten und abstrakten Entitäten des Produktmodells erlaubt, wodurch eine allgemeine Beschreibung unserer Technik möglich wird.

Zusätzlich werden wir eine mögliche Abbildung der im letzten Kapitel eingeführten Aktionssprache AL++ auf die Programmiersprache Java aufzeigen und auf Details zur Erstellung unserer Automatisierungswerkzeuge eingehen.

Am Ende dieses Kapitels werden wir diese Technik dann mit anderen existierenden Techniken vergleichen und Randbereiche vorstellen, die sich mehr oder weniger mit der Automatisierung von Software-Entwicklungsaktivitäten beschäftigen.

Beginnen wollen wir zunächst aber mit der Diskussion eines wichtigen Punktes unserer Technik, dem dokumentzentrierten Charakter des Ansatzes.

5.1 Dokumentzentrierter Ansatz

Zur Speicherung und Bearbeitung von Software-Modellen gibt es prinzipiell zwei „entgegengesetzte“ Ansätze. Der eine Ansatz basiert auf der Nutzung von voneinander „entkoppelten“ Entwicklungsdokumenten, welche jeweils Teile der gesamten Entwicklungsinformation beinhalten. Diesen *dokumentzentrierten Ansatz* findet man i.d.R. bei Programmentwicklungen, die auf Quellcode-Artefakten basieren. So wird man zur „traditionellen“ Entwicklung einer komplexen Java- oder C++-Applikation eine Menge von Source-Code-Dokumenten vorfinden, welche im Quelltext die eigentlichen Entwicklungsinformationen beinhalten. Aber auch Modellierungswerkzeuge wie die Tau SDL Suite von Telelogic (siehe z.B. [Dol03] oder [LEH00]) für die Modellierungssprache SDL [ITU99] nutzen diesen Ansatz.

Der andere Ansatz arbeitet auf einer zentralen Datenstruktur, dem *Repository*, welches sämtliche Daten der Entwicklung speichert (siehe dazu z.B. [Eic91]). Diesen Ansatz findet man typischerweise bei UML-basierten Entwicklungswerkzeugen (in Abschnitt 5.3.3 auf Seite 122 werden wir eine Übersicht über diese Werkzeuge geben). Ihren Ursprung fanden solche Repositorys allerdings bereits in den 1970er Jahren in der Form von „Data-Dictionary“-Systemen (siehe [Stü91]), welche insbesondere in der betrieblichen Informationsverarbeitung noch heute eine wichtige Rolle spielen [Eic91].

Es scheint nun fast so, als würde sich ein *repository-zentrierter Ansatz* idealerweise für Sprachen eignen, die durch Metamodelle beschrieben werden. Wir wollen diese Tatsache an dieser Stelle aber genauer hinterfragen und stellen im Folgenden die Vor- und Nachteile beider Ansätze gegenüber und begründen damit, warum unsere Wahl auf den dokumentzentrierten Ansatz fällt.

Parsen

Bei dem dokumentzentrierten Ansatz ist, wie auch bei Quellcode-Compilern, ein Parsen der Dokumente nötig, um zu einer internen Repräsentation zu gelangen, die für ein Werkzeug zugänglich ist (siehe Parsebäume in Abschnitt 2.1.1). Auch wenn die abstrakte Syntax der Entwicklungsartefakte (und damit der Dokumente oder deren Teile) durch ein OO-Modell beschrieben wird, reicht dies i.d.R. nicht aus. Zur Erstellung der Parser benötigt man eine Definition der konkreten Syntax, weshalb zusätzlich die Erstellung einer Grammatik, welche die lineare Struktur von Textdokumenten bzw. die zweidimensionale Struktur von grafischen Dokumenten widerspiegelt, notwendig wird. Desweiteren ist natürlich auch der umgekehrte Schritt,

also ein „Unparsen“ notwendig, um aus der internen Datenstruktur wieder die konkreten Dokumente zu erhalten.

Bei den repository-basierten Ansätzen entfallen die Schritte des Parsens und des Unparsens, da man direkt auf der internen Datenstruktur arbeiten kann. Natürlich sollte man nicht vergessen, dass ein Werkzeug, das eine Manipulation des Repositorys ermöglicht, auch mit einem gewissen Aufwand zu erstellen wäre (insbesondere wenn es sich wie z. B. bei der UML um grafische Modelle handelt). Desweiteren müssen Vorkehrungen zur persistenten Speicherung der Modellinformation des Repositorys (z. B. mittels einer Datenbank) getroffen werden und entsprechende Schnittstellen zur Verfügung gestellt werden.

Konsistenz

Ein Nachteil bei der Arbeit auf Dokumenten ist, dass an vielen Stellen Inkonsistenzen auftreten können. Das liegt daran, dass die Dokumente, welche redundante Informationen beinhalten können, im Normalfall voneinander entkoppelt sind und eine Konsistenzprüfung (wenn überhaupt) nur zu festgelegten Zeitpunkten erfolgt (z. B. durch Inspektionen). Gerade dieser scheinbare Nachteil ist bei einer genaueren Betrachtung allerdings ein immenser Vorteil, da er die unabhängige Arbeit von Entwicklern und dadurch eine Parallelarbeit zulässt. Oftmals werden Inkonsistenzen nämlich nur kurzzeitig eingeführt, und später wieder korrigiert. So könnte es sein, dass man zu Testzwecken in den lokalen Dokumenten den Typ einer Variable ändert oder die Aufrufparameter einer Methode variiert.

Auch Nissen und Jarke kritisieren in [NiJ99] das Forcieren einer globalen Konsistenz, da dies die „Natur von Konflikten als produktiver Quelle für Kreativität vernachlässigt“ [NiJ99, S. 131]. Ebenso sehen Finkelstein et al. in [FGH94] dieses Problem und gehen sogar soweit zu behaupten, dass „ein Forcieren von Konsistenz den Entwicklungsprozess einschränkt und Neuerungen und Erfindungen erstickt“ [FGH94, S. 571]. Die Autoren schlagen daher vor, Konsistenz nur zu gewissen Zeitpunkten in der Entwicklung zu prüfen und nicht zu erzwingen. Ebenso sehen Liu et al. in [LEM02] die wichtige Aufgabe einer „Unterstützungsumgebung nicht in dem Versuch Inkonsistenzen zu vermeiden, sondern in einer flexiblen Art und Weise wie Inkonsistenzen dem Entwickler angezeigt werden“, insbesondere „weil Beobachtungen von Software-Entwicklern darauf hinweisen, dass diese Inkonsistenzen in den Modellen tolerieren, da die Alternativen das Weglassen von Information oder das treffen verfrühter Entwicklungsentscheidungen wären“ [LEM02, S. 106f.]. Auch gibt es Entwicklungsaktivitäten, die ohne eine inkonsistente Darstellung als Zwischenschritt, nicht realisierbar wären.

Würde man ein repository-basiertes Werzkeug einsetzen, das nur konsistente Änderungen zulässt, müsste man zunächst bei der Einführung einer temporären Inkonsistenz alle betroffenen Stellen ändern. Dies führt zu einem Mehraufwand in der

Entwicklung, da evtl. auch eine mehrfache Benachrichtigung anderer Entwickler notwendig wird und bedeutet, dass unter Umständen viel Arbeit in Änderungen und Korrekturen von sowieso hinfälligen Teilen investiert werden würde.

Lässt man Inkonsistenzen zu – was prinzipiell auch bei Einsatz eines Repositorys möglich ist –, dann sollte man bedenken, dass dies einen Mehraufwand bei einer Werkzeugentwicklung darstellt. Man braucht zusätzliche Mechanismen, welche Konsistenzprüfungen (und evtl. -korrekturen) durchführen. Diese Mechanismen können unserer Erfahrung nach ein erhebliches Ausmaß annehmen (siehe dazu auch die Abschnitte 8.1 und 10.2.3). Im Falle des dokumentzentrierten Ansatzes könnten solche Konsistenzprüfungen Teil des Parsers sein, bei einem Repository müssten redundante oder inkonsistente Informationen identifiziert und aufgelöst werden. Ein Repository-Ansatz, der keine Inkonsistenzen erlaubt, ist natürlich sehr einfach zu realisieren, da Inkonsistenzen vor einer Änderung am Repository überprüft werden können.

Das Zulassen von Inkonsistenzen ist wegen der obigen Vorteile also sinnvoll. Dann sollte allerdings die Dauer zwischen den Zeitpunkten, zu denen solche Konsistenzprüfungen durchgeführt werden, nicht beliebig groß werden, denn sonst könnte die Liste dieser Inkonsistenzen unüberschaubar lang und eine Herstellung von Konsistenz sehr schwierig werden. An dieser Stelle denken ich, dass man sich an den Praktiken gängiger Programmiermethoden in großen Teams orientieren kann. Dort wird zu festgelegten Zeitpunkten (i.d.R. vor dem Einspielen des Codes in die Versionsverwaltung) mindestens die Kompilierbarkeit (was auf Modellen so etwa der Konsistenz zwischen den Dokumenten und deren Metamodell entspricht) gefordert.

Versionierung

Bleibt man bei dem Vergleich zu Programmiertechniken, dann erlaubt der dokumentenzentrierte Ansatz auch die von dort bekannte Versionierung der Modellierungsdokumente. Ein bekanntes Werkzeug, das man zu solchen Zwecken einsetzen kann, ist das *Concurrent Versioning System* (*CVS* [Ber90]). Liegen die Dokumente in einer textuellen Form vor, so lassen sich sogar Differenzen zwischen verschiedenen Versionen anzeigen und so Änderungen feingranular nachvollziehen. Außerdem bietet eine oftmals vorhandene Versionierungshistorie eine Verfolgbarkeit auch auf Prozessebene (siehe z.B. [Que02, 86ff.]).

Bei einer Versionierung eines Repositorys kann man normalerweise nicht auf solche Werkzeuge zurückgreifen. Man kann natürlich prinzipiell das gesamte Repository (z.B. mit Hilfe eines Austauschformats) in eine Versionsverwaltung geben. Dabei erhält man dann allerdings eine äußerst grobgranulare Versionierung, die quasi nur „Schnappschüsse“ des Gesamtprodukts versionieren kann. Dies bedeutet also,

dass bei dem repository-zentrierten Ansatz die Versionsverwaltung integrierter Bestandteil des Repositorys sein muss (siehe dazu auch [Eic91, S.6]).

Einsatz existierender Werkzeuge

Ein letzter Vorteil des Einsatzes von Dokumenten ist, dass Dokumente oft als Eingabe in mehr als ein Werkzeug dienen können. So kann man z.B. ein Quellcode-Dokument als Eingabe für einen Texteditor, für eine integrierte Entwicklungsumgebung (IDE), für einen Debugger und auch für einen Compiler dienen.

Unter einem Betriebssystem wie UNIX kann man desweiteren auf äußerst mächtige Shell-Kommandos zur Bearbeitung und Analyse von Dokumenten zurückgreifen. So ließe sich zum Beispiel mit `awk`, einem Werkzeug zur Textmusterverarbeitung [Dre93, S.303ff.], systematisch alle Vorkommen eines bestimmten Bezeichners gegen einen anderen Identifier austauschen. Damit kann man also bereits mit sehr einfachen Mitteln ein gewisses Maß an Automatisierung auf einer relativ niedrigen Ebene erreichen.

Im Falle von Modelldokumenten macht sich dieser Vorteil verstärkt in der Tatsache bemerkbar, dass für unsere Dokumente bereits sehr mächtige Werkzeuge existieren, die wir für unsere Entwicklungsmethode einsetzen können. So werden wir zur Erstellung der SDL-Dokumente unserer Entwicklungsmethode PROBAnD (siehe Kapitel 6) z.B. auf die oben bereits erwähnte SDL-Entwicklungssuite von Telelogic zurückgreifen.

Will man diese Möglichkeit auch für ein Repository nutzen, dann setzt dies die Einigung auf ein Austauschformat voraus. Für die UML hat man sich z.B. auf das XMI Format (*XML Metadata Interchange* [OMG03a], siehe Abschnitt 5.3.1 auf Seite 118) geeinigt.

Diskussion

Wie man an Hand dieser Punkte erkennt, bieten beide Ansätze sowohl Vor- als auch Nachteile. Man kann einen wichtigen Nachteil des repository-zentrierten Ansatzes ausgleichen, indem man inkonsistente Repositorys zulässt, was dann zu denselben Problemen der Konsistenzprüfung wie bei Dokumenten führt. Umgekehrt kann man eine Art repository-zentriertes Vorgehen forcieren, indem man nach jeder Entwicklungsaktivität, das auf Dokumenten basiert, eine Konsistenzprüfung durchführt, und die Wiederherstellung der Konsistenz fordert.

In diesem Sinne ist ein dokumentzentrierter Ansatz, der nach n Schritten eine Konsistenzprüfung vorsieht, der allgemeinere. Aus diesem Grund und wegen der weiteren oben genannten Vorteile entscheiden wir uns daher für diesen Ansatz. Wie eine Konsistenzprüfung konkret durchgeführt werden kann wird in Abschnitt 8.1 vorgestellt. Im folgenden Abschnitt wollen wir zunächst ein Metamodell einführen,

welches die Typen aller Artefakte in unserem dokumentzentrierten Ansatz beschreibt.

5.1.1 Produktmetamodell

Die Konzepte und Techniken, die den Kern unseres Automatisierungsansatzes darstellen, sind in gewissen Grenzen allgemeingültig und lassen sich daher auf verschiedene Ausprägungen von Entwicklungsmethoden und dadurch auch auf verschiedene Produktmodelle anwenden. Daher wollen wir bei der Beschreibung dieser Technik zunächst von echten Produktmodellen abstrahieren und eine Beschreibung der Typen von Produktmodellentitäten vornehmen, für welche unsere Technik anwendbar ist.

Dazu führen wir, wie wir es in Kapitel 3 schon des Öfteren vorgenommen hatten, eine zusätzliche Metaebene ein. Wir wollen das Modell auf dieser Ebene *Produktmetamodell* nennen. In Abb. 5-1 ist die Einordnung dieses Produktmetamodells in die Modellhierarchie unseres Ansatzes gezeigt.

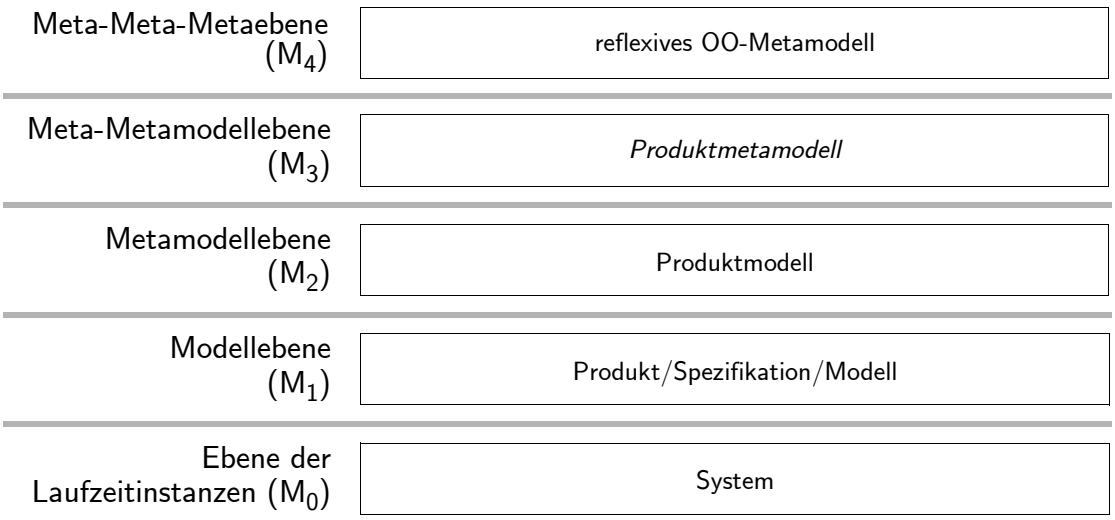


Abbildung 5-1. Beschreibung der Entwicklungsartefakte auf unterschiedlichen Modellebenen

Nach der Definition eines solchen Produktmetamodells lassen sich die betrachteten Produktmodelle (Ebene M_2) als Instanziierung des M_3-Modells verstehen. Jede M_2-Entität entspricht damit einem konkreten Typ eines Entwicklungsartefakts für eine gegebene Entwicklungsmethode.

Wie man in Abb. 5-1 desweiteren erkennt, ist das Produktmetamodell auf Ebene M_3 angesiedelt. Wir bekommen also – im Vergleich zu dem Grafikbeispiel aus dem Kapitel 3 – eine zusätzliche Ebene, bevor wir die Metahierarchie mit einer reflexiven Definition der obersten Ebene abschließen. Für die Definition des Modells auf Ebene M_4 verweisen wir auf Abschnitt 4.1.3.

Wie bereits in Abschnitt 2.1 auf Seite 9 eingeführt wurde, besteht der Formalismus eines Modells aus abstrakten und konkreten Syntaxelementen und einer entsprechenden Semantik. Die Semantik der Entitäten des Produktmetamodells wollen wir – wie schon erwähnt – natürlichsprachlich angeben. Für die abstrakte Syntax wählen wir passenderweise ein OO-Modell.

Eine Definition der konkreten Syntax ist an dieser Stelle eigentlich nicht notwendig, da die Instanzen des Produktmetamodells keine wirkliche physikalische Repräsentation besitzen (es sind nur abstrakte OO-Modelle). Erst auf Ebene M_1 manifestiert sich eine konkrete Syntax in den betrachteten Entwicklungsdokumenten (dort wird dann z.B. die Wahl auf eine SDL-Syntax fallen). Dennoch werden wir eine Klassifikation der konkreten Syntaxelemente vornehmen, da uns dies die allgemeine Beschreibung der Automatisierungstechnik auf der abstrakten Ebene der Produktmetamodelle erlaubt. (Anders als Queins, der in [Que02, S.61ff] auch eine Einordnung verschiedener Entwicklungsartefakte vornimmt, entschlossen wir uns für die Einführung einer Metaebene an Stelle einer Generalisierungshierarchie. Die Gründe für eine Metamodellierung hatten wir bereits ausreichend dargelegt und verweisen daher auf die Kapitel 3 und 4.)

Abstrakte Syntax

In den betrachteten Produktmodellen (und dadurch Produkten) lassen sich zwei Arten von *Artefakttypen* identifizieren. Solche, die bei der gegebenen Betrachtung *atomar* (also unteilbar) erscheinen und solche, die aus weniger komplexen Typen von Artefakten zusammengesetzt sind. In dem Produktmetamodell in Abb. 5-2 sind daher die beiden Objekttypen AtomicArtefactType und ComplexArtefactType als Spezialisierung von ArtefactType definiert. Der ComplexArtefactType weist eine Potenz von (nur) zwei auf, da dieser Typ ausschließlich genutzt wird, um die atomaren Artefakte eines Modells (M_1) zusammenzufassen (siehe auch Abschnitt 5.1.2). (Im Gegensatz zu der Arbeit von Queins [Que02] wählten wir an dieser Stelle bewusst nicht den Begriff „Feature“ für atomare Artefakte, da dieser mit dem „Feature“-Begriff, wie wir ihn in Abschnitt 8.2 auf Seite 208 verwenden werden kollidiert.)

Der ArtefactType beschreibt die gemeinsamen Eigenschaften aller in der Entwicklung entstehenden Artefakte. Neben einem „normalen“ Attribut name zur Benennung der M_2-Instanzen und damit der Entitäten des jeweilig instanziierten Produktmodells besitzt ArtefactType auch Attribute der Potenz zwei. Diese dienen zur Benennung der M_1-Instanzen der Modellelemente (name) und zur Beschreibung oder Spezifikation der Artefakte (description). (Im Vergleich zu Queins (siehe [Que02, S.65]) sehen wir an dieser Stelle keine explizite Versionierung einzelner Artefakte vor (z.B. durch Definition einer Versionsrelation), sondern realisieren eine

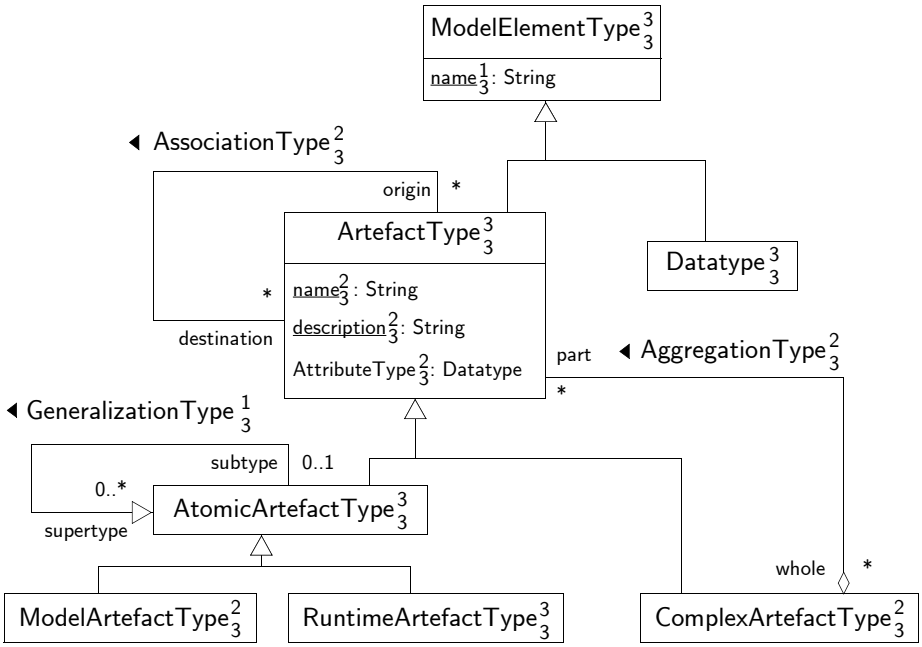


Abbildung 5-2. Klassifikation der abstrakten Artefakte (M_3-Ebene)

etwas grobere Versionierung auf Basis der Entwicklungsdokumente durch ein bereits existierendes Versionierungssystem, siehe Abschnitt 5.1).

Zwischen Instanzen von ArtefactType dürfen Assoziationen definiert werden, die vom Typ AssociationType sind. Da es sich hierbei um einen Typ der Potenz zwei handelt, definiert man damit „echte“ Assoziationen auf der Ebene M_2. In unseren lauffähigen Systemen (Ebene M_0) werden keine Links existieren, weshalb es an dieser Stelle genügt eine Potenz von „nur“ zwei zu wählen. Dasselbe gilt für den Aggregationsrelationstyp zwischen ComplexArtefactType und ArtefactType.

Schließlich erlauben wir noch die Spezialisierung von atomaren Artefakten durch die Einführung einer Generalisierungsrelation auf AtomicArtefactType. Diese hat die Potenz eins, weshalb die eigentlichen Generalisierungsbeziehungen auf Ebene M_2 erscheinen.

Zusätzlich zu Assoziationen, lassen wir auch die Definition neuer Attribute für Artefakttypen zu, indem wir den explizit zu instanziierenden Attributtyp AttributeType deklarieren. Allerdings sind nur Datentypen als Typ der Attributwerte und keine allgemeinen Artefakttypen erlaubt.

Zuletzt unterscheiden wir die atomaren Artefakttypen noch in solche, deren Instanzen nur in der Spezifikation des Software-Systems (Ebene M_1) zu finden sind (ModelArtefactType) und solche, die auch im lauffähigen Produkt auftauchen werden (RuntimeArtefactType). Ein Beispiel für eine Instanz eines ModelArtefactTypes wäre ein M_2-Typ Requirement, da sich die Anforderungen im laufenden System nicht als eigenständige Instanzen manifestieren. Ein Beispiel für eine Instanz eines

RuntimeArtefactTypes wäre eine Instanz ObjectType, deren Instanzen auf M_0 die zur Laufzeit existierende Objekte wären.

Zur Veranschaulichung des Produktmetamodells sind in Abb. 5-3 ein einfaches Produktmodell (als Instanziierung unseres Produktmetamodells) und eine mögliche Instanziierung dieses Modells gezeigt.

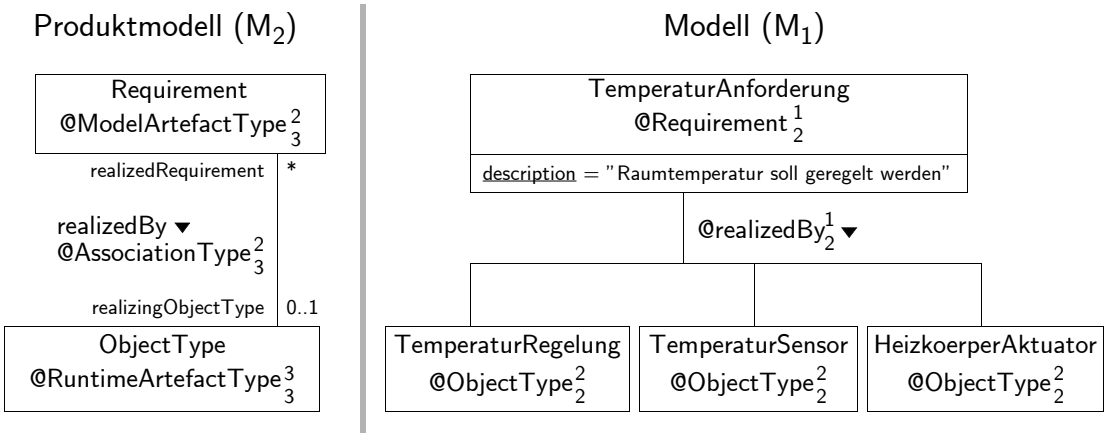


Abbildung 5-3. Beispielhaftes Produktmodell und dessen Instanziierung (abstrakte Artefakte)

Die Instanziierung des Produktmodells beschreibt dabei eine sehr einfache Temperaturregelung, die abhängig von der (Raum-)Temperatur einen Heizkörper ansteuert. Zur Laufzeit werden daher jeweils eine Instanz der TemperaturRegelung, des TemperaturSensors und des HeizkoerperAktuators existieren. Ein weitaus komplexeres Beispiel aus dieser Anwendungsdomäne werden wir in Abschnitt 6.1.3 auf Seite 138 vorstellen.

Konkrete Syntax

Wie bereits erwähnt wurde, braucht es auf der Ebene M_3 noch keine Beschreibung der konkreten Syntax, da es sich bei den Entwicklungsdokumenten um M_1-Instanzen handelt. Wie wir weiter unten aber sehen werden, ermöglicht eine solche „abstrakte“ Definition der konkreten Syntaxelemente eine Beschreibung der Automatisierungstechnik schon auf der Ebene M_3.

Die konkrete Syntax der Entwicklungsdokumente werden wir mit Grammatiken spezifizieren, welche – wie wir bereits in Abschnitt 2.1.2 auf Seite 13 festgestellt hatten – bestens für eine solche Spezifikation geeignet sind.

Für die Klassifikation der konkreten Syntaxelemente brauchen wir allerdings keine Grammatik zu Bemühen, da für abstrakte Elemente ein Metamodell besser geeignet ist. In Abb. 5-4 ist das OO-Modell, welches einer solchen Klassifikation der konkreten Syntaxelemente dient, gezeigt. Jede Instanz einer Modellentität der Ebene M_3 entspricht einem konkreten Syntaxelement auf Ebene M_2.

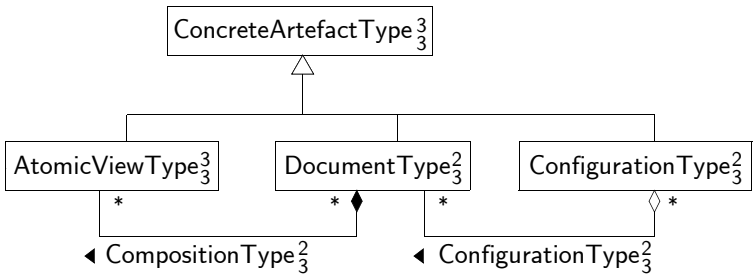


Abbildung 5-4. Klassifikation der konkreten Artefakte (M_3-Ebene)

Beginnen wir mit dem AtomicViewType. Eine Instanz dieses Typs entspricht der konkreten Syntax (also Notation) einer Instanz des Typs AtomicArtefactType aus Abb. 5-2. Wie diese Syntax genau aussieht wird dabei erst auf Ebene M_2 beschrieben. In dieser Arbeit werden wir dies bei der Einführung des Produktmodells für die PROBAnD-Methode in Kapitel 6 vornehmen.

Ein Entwicklungsdokument, das von einem Software-Entwickler oder einem Werkzeug bearbeitet wird, ist aus solchen atomaren *Sichten* („*Views*“) zusammengesetzt (siehe dazu auch [Que02, S.66ff.]). Das bedeutet, dass eine Kompositionsbeziehung zwischen Dokumenttypen (DocumentTypes) und diesen Sichten besteht. Dabei wurde bewusst die Komposition als stärkste Form der Aggregation gewählt, da die Views immer nur zusammen mit ihrem Dokument existieren können (die Lebenszeiten fallen zusammen, siehe Abschnitt 3.1.1 auf Seite 31). Es sei zu beachten, dass es sich hierbei wieder um einen Assoziationstyp der Potenz zwei handelt. Dies bedeutet, dass dieser Assoziationstyp erst für die konkreten Dokumenttypen (auf Ebene M_2) zu einer „echten“ Komposition instanziiert wird. Dann ergibt sich auch aus der Definition der Komposition automatisch eine Multiplizität von eins auf der „Ganzes“-Seite.

Eine letzte Entität, die etwas über die Beschreibung einer konkretem Syntax hinausgeht, ist der ConfigurationType. Diesen führen wir ein, um die Tatsache beschreiben zu können, dass Dokumente zu komplexeren Einheiten, also Konfigurationen, zusammengefasst werden können. So wäre z.B. die Menge aller Dokumente eines Entwicklungsprojekts als entsprechende Konfiguration zu verstehen.

Im Zusammenhang mit den abstrakten Artefakttypen ist sowohl der DocumentType als auch der ConfigurationType eine konkrete Repräsentation des ComplexArtefactTypes. Je nachdem ob nur atomare Views oder Dokumente aggregiert werden, wird in DocumentType oder ConfigurationType unterschieden. So wie dieser ComplexArtefactType sind demzufolge auch der Document- und ConfigurationType Typen der Potenz zwei (Dokumente und deren Konfiguration existieren nur auf Modellebene M_1).

Für das obige Minimalbeispiel würden die konkreten Artefakttypen und deren Instanziierung wie in Abb. 5-5 gezeigt aussehen. Dabei entspricht RequirementView

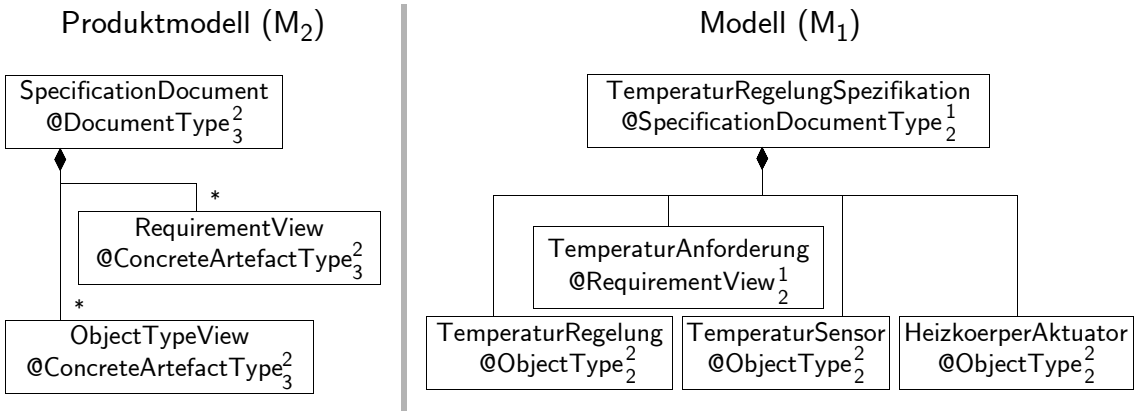


Abbildung 5-5. Beispielhaftes Produktmodell und dessen Instanziierung (konkrete Artefakte)

und ObjectTypeView der konkreten Darstellung der abstrakten Artefakte Requirement und ObjectType. Ein Dokumenttyp, der diese Entwicklungsinformation beinhaltet ist das SpecificationDocument.

In Abb. 5-6 ist das Dokument TemperaturRegelungSpezifikation in einer möglichen konkreten Syntax gezeigt (wir überlassen es dem Leser sich eine passende lineare Grammatik zu überlegen).

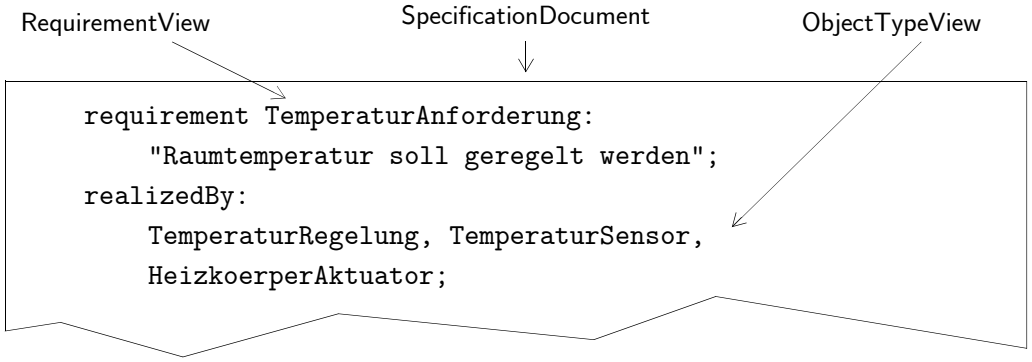


Abbildung 5-6. Ein konkretes Dokument

An diesem Beispiel sieht man auch sehr schön, wie sich eine Assoziation zwischen abstrakten Artefakten (realizedBy) in einem konkreten Dokument manifestiert. Wegen der linearen Struktur der Dokumente ist hier natürlich eine Art von Verweis auf die entsprechend verbundene Instanz notwendig.

5.1.2 Modelltransformationen

Mit den oben eingeführten M_3-Typen kann man nun allgemeingültig die Vorgehensweise für unsere Modelltransformationen beschreiben. In einem dokumentzentrierten Ansatz setzt sich jede solcher Modelltransformationen aus den folgenden Ab-

bildungsschritten zusammen (wir hatten diese Kette von Abbildungen bereits in Abschnitt 2.2.2 auf Seite 19 vorgestellt):

$$M_{\mathrm{e}} \xrightarrow{\mathrm{0F}} M_{\mathrm{e,\,i}} \xrightarrow{\mathrm{M0}} M_{\mathrm{a,\,i}} \xrightarrow{\mathrm{0F}} M_{\mathrm{a}}$$

Dabei sind M_{e} und M_{a} die Eingabe- bzw. Ausgabedokumente auf Ebene M_1. Diese sind also durch zweifache Instanziierung des Produktmetamodells entstanden. (Durch die Instanziierung des Produktmetamodels erhalten wir ein Produktmodell, dessen Instanzen wiederum die konkreten Entwicklungsartefakte darstellen.)

Bei $M_{\mathrm{e,\,i}}$ und $M_{\mathrm{a,\,i}}$ handelt es sich um die Modelle in einer internen Darstellung (Datenstruktur). Die konkrete Syntax einer solchen internen Repräsentation anzugeben fällt schwer, da diese oft mit den abstrakten Konzepten zusammenfällt. Als Beispiel sei hier der abstrakte Syntaxbaum eines Hochsprachen-Compilers genannt, den wir bereits in Abschnitt 2.1.1 auf Seite 11 als Beispiel zitiert hatten.

Analog zu dem Compiler-Beispiel würde man auch die erste Transformation als „Parsen“ und die letzte Transformation als „Unparsen“ bezeichnen, weshalb beide vom Typ 0F sind. Bei der Transformation in der Mitte handelt es sich dann um die eigentliche das Modell ändernde Transformation und diese ist daher vom Typ M0.

In Abb. 5-7 sind diese elementaren Arten der Modelltransformation im Kontext der oben eingeführten Begriffe gezeigt. Dabei handelt es sich um eine Darstellung auf der M_1-Ebene, weil erst die Entitäten dieser Ebene wirklich transformiert werden.

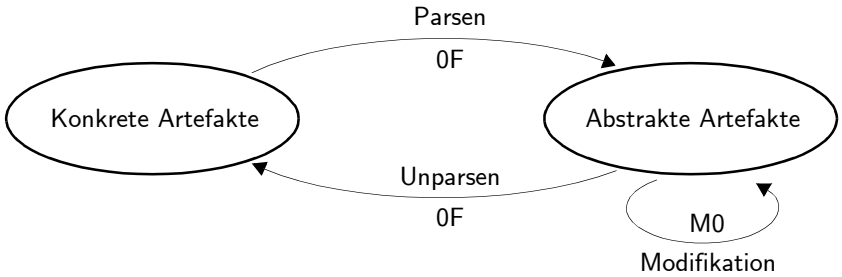


Abbildung 5-7. Einordnung von Transformationen (M_1-Ebene)

Wir wählten in dieser Abbildung bewusst die Mengennotation für die Menge der konkreten und abstrakten Artefakte. Interpretieren wir diese wie in Abschnitt 3.1 auf Seite 30 als Extension eines Typs, gelangen wir bereits zur Ebene M_2. Durch eine weitere Abstraktion erhielten wir dann schließlich die ConcreteArtefactTypes und ArtefactTypes unseres Produktmetamodells (Ebene M_3). Nutzen wir unsere Instanziierungsnotation aus Kapitel 3, so kann man ein M_1-Element e auch als zweifache Instanziierung eines M_3-Elements MT beschreiben als

$$e@@MT. \qquad \textbf{(Gleichung 5-1)}$$

Diese Notation wollen wir nun in den folgenden Abschnitten zur allgemeingültigen Beschreibung unserer Automatisierungsansätze verwenden.

Parsen

Das Schema der Parse-Vorgänge, also die Abbildung von konkreten auf abstrakte Artefakte, ist in Abb. 5-8 gezeigt.

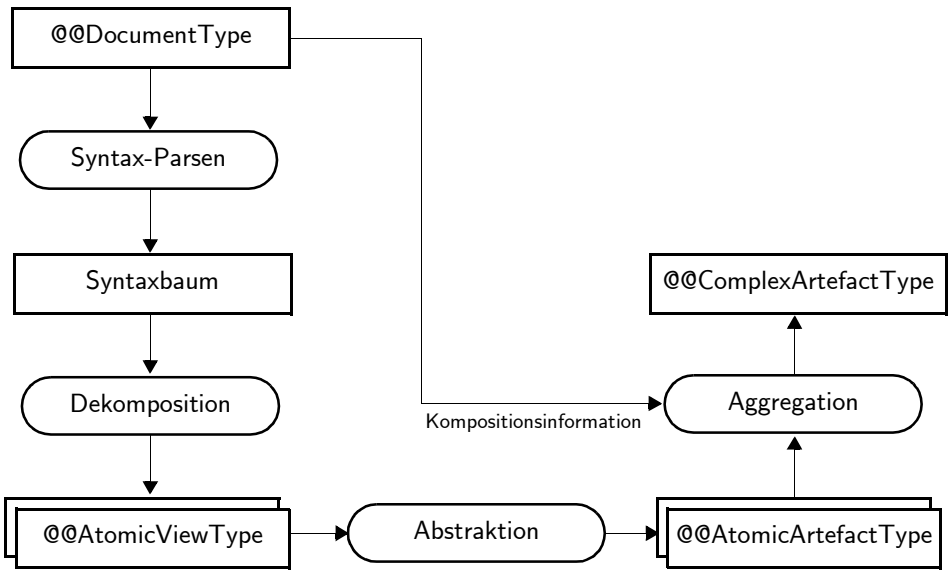


Abbildung 5-8. Systematisches Parsen von Dokumenten

Die konkrete Syntax eines Dokuments ist mittels einer Grammatik spezifiziert. Daher ist der logische erste Schritt in dem gesamten Parse-Vorgang einen entsprechenden Syntaxbaum aufzubauen. In Abschnitt 2.1.2 hatten wir bereits existierende Parser-Generatoren erwähnt, welche Code für eine solche Aktivität automatisch erzeugen können. Diese lassen sich an dieser Stelle sinnvoll und effizient einsetzen.

Ausgehend von dem erzeugten Syntaxbaum führt man dann eine Zerlegung (oder Dekomposition) des Eingabedokuments in dessen atomare Views durch. Wie in Gl. 5-1 beschrieben, definiert man mit @@DocumentType eine konkrete Dokumentinstanz und mit @@AtomicViewType eine konkrete Instanz eines Views.

Als Nächstes muss evtl. von der konkreten Syntax einzelner Views abstrahiert werden und die „eigentliche“ Entwicklungsinformation extrahiert werden. Das Ergebnis dieser Abstraktionsaktivität ist eine Menge abstrakter atomarer Artefakte. Dieser Abstraktionsschritt ist – wie auch das ursprüngliche Erstellen des Syntaxbaums – abhängig von der gewählten konkreten Syntax und kann daher nicht allgemeingültig beschrieben werden.

Im Beispiel aus Abb. 5-6 würde dies z.B. für das RequirementsView die Zerlegung in einen name-Teil (vor dem „:“) und einem description-Teil (nach dem „:“) erfordern. Wenn man die Grammatik feiner beschreibt, also Name und Beschreibung als

Nichtterminale aufnimmt, ist dieser Schritt sogar noch einfacher, denn man muss nur den Text entsprechend speichern.

Ein aufwändiger Schritt bei dieser Abstraktion ist allerdings die Überprüfung, ob ein konkretes Artefakt bereits geparsed wurde, das dem aktuell betrachteten Artefakt entspricht. In unserem kleinen Beispiel aus Abb. 5-5 wäre dies z.B. der Fall, wenn ein identischer TemperaturAnforderung-View auch in einem anderen Dokument (z.B. der Problembeschreibung) auftauchen würde. Diese Situation ist charakteristisch für unseren dokumentzentrierten Ansatz, da Informationen in mehr als einem Dokument gleichzeitig abgelegt werden können.

In einer solchen Situation darf als Ergebnis der Abstraktion nicht ein neues abstraktes Artefakt erzeugt werden, sondern die bereits erzeugte Instanz des AbstractArtefactType muss referenziert werden. Dies ist notwendig, da sonst die Verfolgbarkeit zwischen den abstrakten Artefakten verloren gehen würde. Desweiteren muss diese Instanz um eventuell neu hinzugekommene Informationen ergänzt werden, was eine kritische Stelle darstellt, da Inkonsistenzen auftauchen können und entsprechend behandelt werden müssen. Wir werden uns in Abschnitt 8.1 auf Seite 188 genauer mit dem Problem solcher Inkonsistenzen beschäftigen.

Um zu einem späteren Zeitpunkt aus diesen Artefakte wieder die gewünschten Dokumente erzeugen zu können, werden die abstrakten Artefakte zu einem komplexen Artefakt aggregiert. Dabei nutzt man die Kenntnis aus der Dekompositionsaktivität, welche Views in dem betrachteten Dokument enthalten waren und welche atomaren Artefakte demzufolge in einem komplexen Artefakte aggregiert werden müssen.

Das Einführen von komplexen Artefakten zur Aggregation von atomaren Artefakten eines Dokuments ist auch deshalb relevant, da man nicht garantieren kann, nur aus den Assoziationen zwischen atomaren Artefakten (siehe Abb. 5-2 auf Seite 88) die notwendigen atomaren Artefakte zu bestimmen, die man für ein Erzeugen eines Dokumentes wieder bräuchte. Insbesondere wenn ein atomares Artefakt keine Assoziation zu einem anderen Artefakt besitzt kann dieses Artefakt nur über die Aggregationsbeziehung, die aus der Dekomposition entsteht, aufgefunden werden.

Werden mehrere Dokumente geparst, dann wird zusätzlich zu den komplexen Artefakten, welche die Komposition der Dokumente aus Views widerspiegeln, weitere komplexe Artefakte erzeugt, welche die Konfiguration der Dokumente zu einer komplexeren Spezifikation beschreiben.

Unparsen

Bei dem Unparse-Schritt handelt es sich prinzipiell um die Rückrichtung des oben beschriebenen Parse-Schritts (siehe auch [MeQ03] und [Met03]). Ein entsprechendes Schema ist in Abb. 5-9 gezeigt.

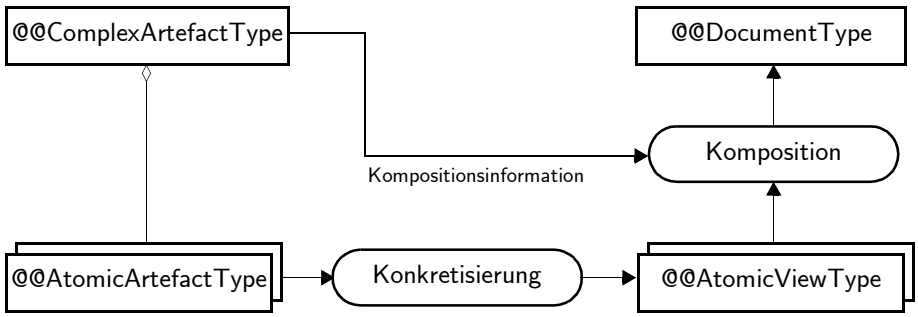


Abbildung 5-9. Systematisches Unparsen von Dokumenten

Zunächst wird für jedes atomare Artefakt eines komplexen Artefakts eine konkrete Repräsentation in der jeweiligen konkreten Syntax erzeugt. Handelt es sich bei dem komplexen Artefakt um eine Aggregation von weiteren (weniger) komplexen Artefakten, dann wird hierarchisch vorgegangen, also der beschriebene Schritt auf die weniger komplexen Artefakte angewendet. Schließlich werden die erzeugten Views zu einem Dokument zusammengesetzt.

Wie beim Parse-Vorgang sind auch hier die Details der Konkretisierungsaktivität von der gewählten konkreten Syntax abhängig und werden daher auf der abstrakten Ebene noch nicht beschrieben. Eine sinnvolle technische Basis einer solchen Aktivität können allerdings Vorlagen („Templates“) darstellen (siehe dazu [MeQ03]).

Modifikation

Die eigentliche Modellmodifikation (zum Zwecke einer Modellevolution) arbeitet schließlich nur auf den abstrakten Syntaxelementen, was eine deutliche Vereinfachung der Beschreibung solcher Aktivitäten bietet, da man von konkreten Notationen abstrahieren kann.

Beispiele für solche Modellmodifikationen folgen im zweiten Teil der Arbeit (ab Kapitel 7).

Abkürzung von Transformationsketten

Eine typische Automatisierung von Entwicklungsaktivitäten wird aus der Hintereinanderausführung vieler kleinerer Aktivitäten bestehen, die jeweils von einem anderen Automatisierungswerkzeug (oder von Hand) ausgeführt werden. In unserem dokumentbasierten Ansatz wird dabei für jede Werkzeuganwendung ein Parse- und Unparse-Schritt notwendig.

Will man nun mehrere Werkzeuganwendungen direkt hintereinander ausführen, dann werden viele solcher Parse-/Unparse-Schritte gebraucht, die nicht wirklich notwendig sind, da die „temporären“ Dokumente nicht von einem Entwickler bearbeitet werden.

Zur Vereinfachung – und dadurch Beschleunigung – solcher Ketten verwenden wir daher ein Austauschformat für die interne Datenstruktur der Werkzeuge, welches von der konkreten Syntax der Entwicklungsdokumente abstrahiert und generisch für alle Instanzen des vorgestellten Produktmetamodells anwendbar ist.

Ein weiterer Vorteil ist, dass sich auch unvollständige Modellinformation austauschen lässt, die sich nicht in den Dokumenten abbilden ließe. Ein Beispiel wären atomare Artefakte, die von keinem komplexen Artefakt aggregiert werden. Mit unserem Parse-/Unparse-Schema würden diese in keinem Dokument auftauchen und daher verschwinden.

Bei den Abbildungen mit Hilfe eines solchen Austauschformats handelt es sich zwar nach wie vor um Ketten von Abbildungen der Typen 0F, M0 und 0F; allerdings ist das Parsen und Unparsen (0F) nun erheblich vereinfacht und erfordert auch keine Änderungen am Austauschformat, wenn sich die konkrete Syntax der Dokumente ändert.

Die technischen Details und die Realisierung dieses Austauschformates werden im Folgenden zusammen mit der technischen Realisierung unserer Automatisierungswerkzeuge erläutert.

5.2 Technische Realisierung

Um zu ausführbaren Automatisierungswerkzeugen zu gelangen, nehmen wir eine Realisierung der Artefakttypen des Produktmodells und der Aktivitäten des Prozessmodells in der Programmiersprache Java vor (eine Einführung in die Sprache findet sich z. B. in [MSS96] oder [Poe00]). Anders als unsere zu Dokumentationszwecken eingeführten Multiebenenmodelle mit Operationen in der Aktionssprache AL++ lässt sich dieser Java-Code direkt auf einer Java Virtual Machine (JVM) zur Ausführung bringen.

Bei einer Abbildung auf ausführbaren Code müssen also sowohl die strukturellen Aspekte, die im Produktmodell beschrieben sind, als auch die Verhaltensaspekte, die zur Transformation von Modellen im Prozessmodell eingeführt wurden, berücksichtigt werden. Grob gesagt bilden wir die strukturellen Aspekte (insbesondere die Objekttypen) auf Java-Klassen ab. Die Verhaltensaspekte werden in Java-Methoden realisiert. Eine Übersicht über diese beiden Aspekte, die in den folgenden Unterabschnitten genauer behandelt werden, zeigt Abb. 5-10.

Eingangs- und Ausgangsprodukte sind normalerweise Dokumente. Bei einer Verkettung von Werkzeugen sind aber auch Dateien in einem Austauschformat (siehe Abschnitt 5.2.4) erlaubt.

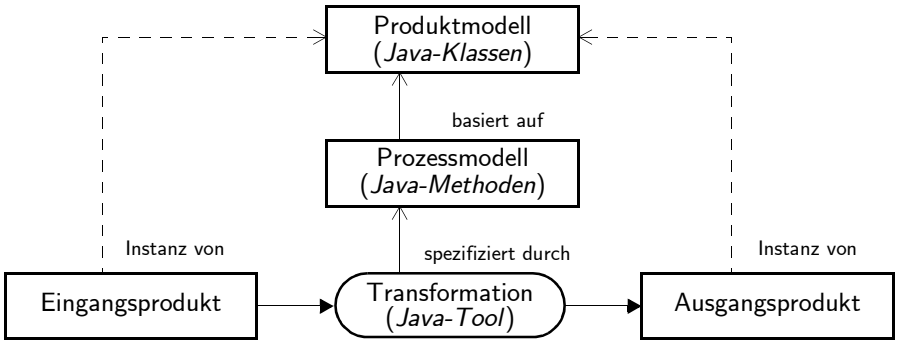


Abbildung 5-10. Realisierung der Modelltransformation (Verfeinerung v. Abb. 2-7 auf Seite 20)

5.2.1 Abbildung von Produktmodellen auf Java-Code

Die für die Abbildung von Produktmodellen auf Code genutzte Eigenschaft der Metamodellierung ist deren „relative“ Interpretationsmöglichkeit. Diese erlaubt, dass man jedes Metamodell auch wieder als Modell interpretieren kann (ein Metamodell ist ein „Modell eines Modells“, siehe Abschnitt 2.1.1 auf Seite 11). Daher ist es möglich, bereits existierende Modellierungstechniken und -werkzeuge, die typischerweise für Modelle der Ebene M_1 konzipiert wurden, auch für Metamodelle einzusetzen. Folglich interpretieren wir unser Produktmodell, das auf der Metamodellebene angesiedelt ist, als objektorientiertes Modell. Da wir unsere Produktmodelle als UML-Klassendiagramme beschreiben können (wir hatten dies z. B. zur Illustration unseres Grafikbeispiels bereits getan), können wir daher UML-Werkzeuge für die Java-Code-Generierung einsetzen. Für unsere Arbeit entschieden wir uns dabei für Rhapsody der Firma iLogix [Dou00, S. 16][GHP02], da dieses Werkzeug unseren Anforderung an den erzeugten Java-Code erfüllt (siehe unten) und darüberhinaus in einer „Modeler“-Edition frei erhältlich war.

Die wichtigsten strukturellen Elemente, die es auf Java-Code abzubilden gilt, sind Objekttypen, Attribute und Relationen.

Wir hatten in Kapitel 3 bereits erwähnt, dass eine Implementierung des Objekttyps die Klasse ist. Nichts liegt also näher als die Objekttypen unseres Produktmodells als Java-Klassen zu realisieren. In diesem Zusammenhang bietet es sich auch an, die Spezialisierungsrelation als Vererbungsrelation zu implementieren, da Java dies als objektorientierte Programmiersprache natürlich anbietet. Da wir für unsere Modelle die einfache Spezialisierung gefordert hatten, ist auch die Abbildung auf Java unproblematisch (Java lässt nur die einfache Vererbung von Klassen zu). Diese Abbildung wird auch standardmäßig von jedem UML-Werkzeug, insbesondere von Rhapsody, unterstützt.

Als Nächstes müssen die Attribute abgebildet werden. Hier bietet sich neben einer öffentlichen Deklaration von Attributen als Felder der zugehörigen Java-Klasse

auch deren Realisierung über private Felder und öffentliche *Zugriffs-* und *Mutatormethoden* an. Letztere Möglichkeit erlaubt eine stärkere Kapselung von Informationen und ein höheres Maß an „Information Hiding“ (siehe dazu das Grundlagenpapier von Parnas et al. [PCW01]).

Die bekannteste Konvention zur Benennung der Zugriffsmethoden ist die der JavaBeans (siehe [ONe98]). Bei einem gegebenen Attribut `meinAttribut` vom Typ `MeinTyp` müssen zwei Methoden mit den folgenden Signaturen existieren:

```
public void setMeinAttribut(MeinTyp p_MeinTyp)
```

zum Setzen des Attributwertes der im Parameter `p_MeinTyp` übergeben wird (Mutatormethode) und

```
public MeinTyp getMeinAttribut()
```

zum Auslesen der aktuellen Attributbelegung (Zugriffsmethode). Der Vollständigkeit halber sei erwähnt, dass im Falle eines Typs `boolean` der Name der Zugriffsmethode mit „`is`“ beginnt.

Auch diese Abbildung von Attributen ist noch so gut wie in jedem UML-Werkzeug zu finden. Bei der Abbildung von Assoziationen ist dies allerdings nicht unbedingt der Fall. Rhapsody bietet solch eine Abbildung jedoch an, was mit ein Grund für dessen Wahl war. Dabei erzeugt Rhapsody Zugriffs- und Mutatormethoden für Rollen, was analog zu dem Assoziationszugriff in unserer Aktionssprache ist (siehe Abschnitt 4.2.2 auf Seite 65).

Um die unterschiedlichen Abbildungen von Rollen auf Java-Methoden zu illustrieren, wollen wir das bereits eingeführte Produktmodellbeispiel heranziehen. In Abb. 5-11 ist dieses Modell nochmals gezeigt.

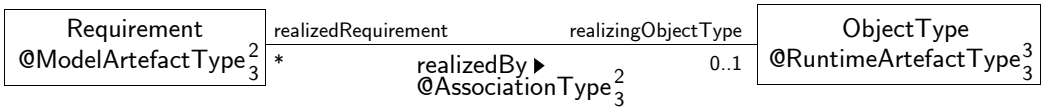


Abbildung 5-11. Modell zur Veranschaulichung der Abbildung von Assoziationen

Aus diesem UML-Klassendiagramm wird (unter anderem) der folgenden Java-Code für die Klasse `Requirement` erzeugt:

```
public class Requirement {
    protected ObjectType realizingObjectType;
    public ObjectType getRealizingObjectType() {
        /* ... */
    }
    public void setRealizingObjectType(ObjectType p_ObjectType) {
        /* ... */
    }
```

```
    /* ... */
}
```

Neben dem Code für die Modifikation der Datenstruktur für das Link-Ende (hier `realizingObjectType`) wird auch automatisch Code erzeugt, der für die korrekte Belegung von bidirektionalen Relationen sorgt. Dieser Code, der ein sog. „bidirectional reference handshaking“ [IyM04] realisiert, ist hier nicht gezeigt.

Etwas anders sieht der Code aus, wenn es sich um Assoziationsenden mit einer Multiplizität größer eins handelt:

```
public class ObjectType {
    protected LinkedList realizedRequirement;
    public ListIterator getRealizedRequirement() {
        /* ... */
    }
    public void addRealizedRequirement(Requirement p_Requirement) {
        /* ... */
    }
    public void removeRealizedRequirement(Requirement p_Requirement) {
        /* ... */
    }
    /* ... */
}
```

In diesem Fall liefert die Zugriffsmethode (`get`) keine Referenz auf ein einzelnes Objekt, sondern eine Referenz auf eine Datenstruktur, welche eine „Menge“ von Objekten beinhaltet. Genauer gesagt liefert Rhapsody einen Iterator (siehe z.B. [Eck02, Kapitel 11]), mit dem sich mittels entsprechender Methoden wie z. B. `next()` systematisch über die Datenstruktur laufen lässt. Die Anwendung eines solchen Iterators werden wir weiter unten zeigen.

Es sei an dieser Stelle angemerkt, dass Rhapsody (zumindest in seinen Standardeinstellungen) von der Interpretation eines Link-Endes als Menge abweicht und stattdessen Listen (also eine geordnete Menge von Elementen, die auch Duplikate beinhalten kann) verwendet. Dies ist auch der Grund, weshalb eine Referenz vom Typ `ListIterator` und nicht vom einfacheren Typ `Iterator` zurückgeliefert wird. Wichtig ist diese Tatsache insofern, als dass man nun als Programmierer dafür sorgen muss, dass keine doppelten Links angelegt werden. Vorteil dieser Implementierung ist allerdings, dass damit auch „geordnete“ Assoziationen, also solche, die das UML-Stereotype `{ordered}` besitzen, sehr einfach realisiert werden können (siehe [RJB99, S. 374ff.]).

Der Name der Mutatormethode beginnt bei Assoziationen mit höheren Multiplizitäten nicht mit `set` sondern mit dem treffenderen `add`. Zusätzlich wird eine Me-

thode zum Entfernen von Objekten aus den Link-Enden erzeugt (`remove`), da dies über die `add`-Methode nicht möglich ist (bei einer Multiplizität von eins würde z.B. der Aufruf `setRealizingObjectType(null)` das Objekt entfernen).

Zuordnung der Verantwortlichkeiten zu Klassen

Nachdem wir nun gesehen haben, wie man allgemein ein Produktmodell auf Java-Code abbilden kann, wollen wir nochmals auf die Klassifikation der Produktmodellentitäten aus Abschnitt 5.1.1 zurückkommen. Diese bieten nämlich einen sehr guten Ansatz für die Aufteilung von Verantwortlichkeiten auf die Klassen, die bei der Abbildung auf Java-Code erzeugt werden.

So ist es offensichtlich, dass für alle Instanzen des ArtefactTypes eine entsprechende Java-Klasse generiert wird, die gleichsam dann auch einen Teil der abstrakten Datenstruktur bildet, auf welchem die Automatisierungswerkzeuge zur Modelltransformation vom Typ M0 arbeiten. Eine solche Modellmodifikation entspricht damit einer Modifikation der Instanzen dieser Java-Klassen.

Bildet man nun aber zusätzlich auch die Instanzen der ConcreteArtefactTypes auf Java-Klassen ab, dann erhält man ein Schema, das die Implementierung der Parser- und Unparser-Werkzeuge erleichtert und auch mit eine wichtige Systematik unseres Ansatzes darstellt. Die Vereinfachung basiert auf der Tatsache, dass ein „Separation of Concerns“ stattfindet, da man nun spezifische Klassen für die Aufgabe des Parsens oder Unparsens vorfindet. Bei diesem Vorgehen gebührt somit die Aufgabe des Parsens eines Dokuments eines bestimmten Typs der entsprechenden Java-Klasse, welche nach einer Zerlegung des Dokuments in dessen Views (Dekompositionsaktivität in Abb. 5-8 auf Seite 93) die Aktivität der Abstraktion von diesen Views auf die Klassen, welche die jeweiligen View-Typen implementieren, delegiert. Die View-Klassen liefern als Rückgabewerte jeweils das zugehörige atomare Artefakt. Diese atomaren Artefakte werden dann wieder in der Dokumenttyp-Klasse aufgesammelt und zu einem entsprechenden komplexen Artefakt zusammengesetzt.

Für das Beispiel aus Abschnitt 5.1.1 würde es also neben den Java-Klassen für die abstrakten Typen von Artefakten (`Requirement` und `ObjectType`) auch Java-Klassen für die atomaren Views (`RequirementView` und `ObjectTypeView`) und eine Java-Klasse für das Dokument (`SpecificationDocument`) geben. In Abb. 5-12 ist die Delegation von Verantwortlichkeiten für die Dokumentinstanz aus Abb. 5-6 auf Seite 91 mit Hilfe eines Sequenzdiagramms (siehe z.B. [JRH03, S.323ff.]) gezeigt.

Zusätzlich zu dem abstrakten Produktmodell aus Abb. 5-3 gehen wir davon aus, dass eine Instanz des ComplexArtefactType mit Namen Specification existiert, welches die atomaren Artefakte des Dokuments entsprechend dem Ansatz aus Abschnitt 5.1.2 aggregiert. Dem aufmerksamen Leser wird auffallen, dass an Stelle der üblichen Instanzen am Kopf des Diagramms Typen eingetragen sind. Dies liegt da-

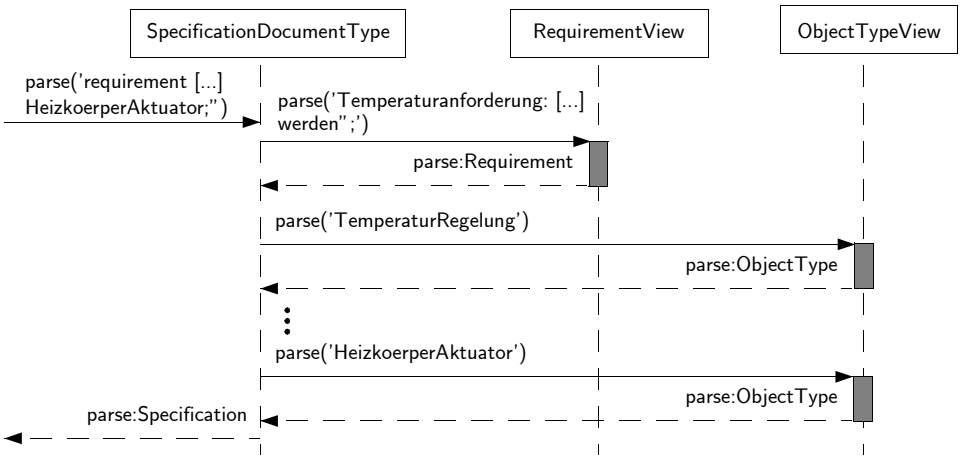


Abbildung 5-12. Delegation von Verantwortlichkeiten auf Klassen

rin begründet, dass die Methoden zum Parsen als sog. statische Methoden definiert sind, die aus dem Klassenkontext aufgerufen werden können (im Zusammenhang mit unserer Multiebenenmodellierung handelt es sich dabei um Operationen der Potenz „0“).

5.2.2 Abbildung von Aktivitäten auf Java-Code

Die Abbildung der Verhaltenskonstrukte der Aktionssprache AL++ auf Java kann nun mit Hilfe der obigen Methoden für den Zugriff auf Attribute und Links geschehen. Für den Zugriff auf Attribute und Links verwendet man offensichtlicherweise die `get`-Methoden, für eine Modifikation die `set-` oder `add`-Methoden, bzw. die `remove`-Methoden zum Entfernen von Links. Zu beachten ist natürlich der unterschiedliche Rückgabewert bei Assoziationsenden mit einer Multiplizität größer eins, da hier anders als bei unserer Aktionssprache (wo ein `Set` geliefert wird), nur ein `Iterator` zurückgegeben wird.

Ein weiteres Konstrukt, das es abzubilden gilt, ist das `foreach`-Konstrukt. Dazu wiederholen wir hier nochmals dessen Syntax:

```
foreach(<objektReferenz> in <mengenReferenz>)
    <anweisungsBlock>
```

Da wir von den Zugriffsmethoden einen Iterator für den Zugriff auf die Link-Enden geliefert bekommen, können wir diesen direkt nutzen, um das `foreach`-Konstrukt zu realisieren. Nehmen wir also an, die `<mengenReferenz>` bezieht sich auf einen Rückgabewert einer solchen Zugriffsmethode, dann liefert diese in unserem Fall eine `<iteratorReferenz>`. Damit kann das folgendes Code-Fragment für obige Aktion verwendet werden:

```
while(<iteratorReferenz>.hasNext()) {
```

```
        <objektReferenz> = (<typAmLinkEnde>)<iteratorReferenz>.next();
        <anweisungsBlock>
    }
```

Mit Hilfe der Iterator-Methoden `hasNext()` und `next()` laufen wir also über alle Elemente der Liste, erzeugen eine korrekt typisierte Referenz durch einen Cast-Operator (siehe z.B. [Eck02, Kapitel 3]) und rufen anschließend die Aktionen im Block `<anweisungsBlock>` auf.

Kritisch an dieser Stelle ist, wenn – während über die Elemente des Link-Endes iteriert wird – eine Änderung der dem Iterator zu Grunde liegenden Menge (mit Hilfe der entsprechenden Mutatormethoden) vorgenommen wird. In Java führt dies direkt zu einer Exception, einem sog. „Concurrent Modification Error“ (siehe z.B. [Eck02, Kapitel 11]). Generell deutet dies auf einen Fehler oder einen kritischen Punkt in der Implementierung hin und daher müssen Maßnahmen (d.h. Änderungen am Code) getroffen werden, um dies zu vermeiden oder zu umgehen.

In unserem Fall hatten wir bereits auf der Ebene der Sprache AL++ gefordert, dass keine Änderungen an der Menge, über die iteriert wird, durchgeführt werden dürfen. Daher kann man dann im obigen Konfliktfall eine intermediäre Menge erzeugen (was zwar die ineffizientere aber direktere Abbildung der Aktionssprache auf Java wäre). Ein Beispiel dazu folgt weiter unten.

Mit Erscheinen der Java 2 Platform Standard Edition (J2SE) Version 1.5 [Aus04] im Frühjahr 2004 lässt sich die obige Abbildung auch einfacher realisieren. Diese Java-Version führt neben dem Mechanismus der „*Generics*“ (typisierte Parameter, siehe Abschnitt 5.3.4 auf Seite 126) auch einen erweiterte Form der `for`-Schleife ein. Damit stellt sich die Abbildung der `foreach`-Schleife wie folgt dar:

```
for(<objektReferenz> : <mengenReferenz>)
    <anweisungsBlock>
```

Voraussetzung ist, dass es sich bei der `<mengenReferenz>` um eine Referenz auf eine typisierte Menge handelt, die in J2SE 1.5 folgendermaßen definiert wird (der Typparameter wird in `<>` eingeschlossen an den Namen des Typs angehängt):

```
Set< <typAmLinkEnde> > <mengenReferenz> = /* ... */;
```

Abbildung der Multiebenenmodellierung

Die Realisierung der in Abschnitt 4.1 auf Seite 55 eingeführten Multiebenenmodellierung in Java ist leider nicht ohne Probleme. Dies liegt hauptsächlich an der fehlenden Unterstützung einer beliebigen Anzahl von Metaebenen in der Programmiersprache. Standardmäßig werden nur die Instanzen- und Klassenebene in vollem Unfang unterstützt.

Eine Metaklassenebene existiert zwar, sie ist aber fix und kann nicht geändert werden. Die wichtigsten Metaklassen, die von Java zur Verfügung gestellt werden sind `Class`, `Method`, `Field` und `Constructor` (siehe dazu auch [Poe00, S.79ff]). Jede Java-Klasse ist eine Instanz der Metaklasse `Class`, entsprechendes gilt für Methoden und Felder.

Mit Hilfe des *Introspektionsmechanismus* der Sprache (siehe [Poe00, S.80]) lässt sich nun mittels dieser Metaebene zur Laufzeit auf beliebige Klasseninformationen zugreifen. So wird z.B. die Abfrage der für eine Klasse definierten Felder mittels der Methode `public Field[] getFields()` von `Class` realisiert. Um die Klasse (also die `Class`-Instanz) einer Java-Instanz zu bestimmen existiert die Methode `getClass()`. Diese erlaubt die direkte Abbildung des `type`-Operators von AL++.

Neben der Feststellung vorhandener Methoden, Felder und Konstruktoren, erlaubt Java mit der Technik der *Reflexion* (engl. „Reflection“, vgl. [CST00, S.2]) auch deren Anwendung. So lassen sich z.B. mit der Methode `newInstance()` von `Class` zur Laufzeit Instanzen einer beliebigen Klasse erzeugen (siehe dazu wieder [Poe00, S.80]). Wie oben schon angedeutet, lässt sich diese Methode allerdings nicht auf die Metaklasse `Class` anwenden, da `Class` keinen Konstruktor besitzt (das Erzeugen neuer Klassen ist allein Aufgabe der Virtuellen Maschine).

Mit diesen Voraussetzungen können die Modellebenen wie in Abb. 5-13 gezeigt nach Java abgebildet werden.

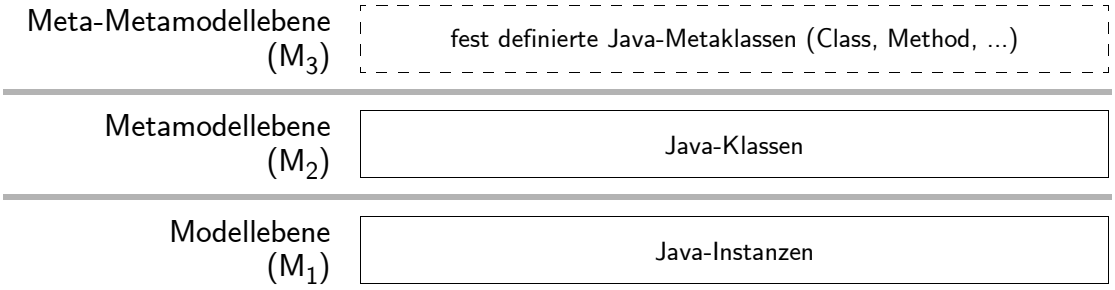


Abbildung 5-13. In Java realisierte Ebenen der Multiebenenmodellierung

Stehen nur diese zwei (bzw. drei) Ebenen zu Verfügung, dann muss man für viele der Probleme, die man elegant mit der verbesserten Multiebenenmodellierung lösen kann, auf alternative Lösungen ausweichen. Wir hatten diese bereits im letzten Kapitel ausführlich diskutiert. Als Beispiel seien hier die Powertypes genannt, die sich durch entsprechende Vererbungsbeziehungen implementieren lassen (siehe z.B. Abb. 3-6 in Abschnitt 3.2.1) oder die Realisierung höherpotenter Relationen durch das Einführen der entsprechenden Objekttypen (siehe z.B. Abb. 5-9 in Abschnitt 3.2.1). Diese Alternativen lassen sich alle direkt auf Java abbilden.

Insbesondere bedeutet die fixe M_3-Ebene, dass wir das Produktmetamodell nicht direkt auf Java abbilden können, sondern diese Ebene nur einer „externen“ Beschrei-

bung dienen kann. In der konkreten Implementierung finden sich daher für viele der Produktmetamodell-Entitäten die entsprechenden Oberklassen auf der Ebene des Produktmodells (Stichwort: „Powertypes“).

Diskussion

An den obigen Abbildungen auf Code wird sicherlich nochmals der Vorteil einer implementierungsunabhängigen Aktionssprache deutlich. Neben der unterschiedlichen Syntax und Semantik der Zugriffs- und Mutatormethoden im Java-Code stellt sich auch das Iterieren über die Elemente einer Menge per Iterator umständlich dar und bietet eventuelle kritische Stellen. Desweiteren hatten wir bereits in Abschnitt 4.1 auf Seite 55 gesehen, wie man durch die Einführung der verbesserten Multiebenenmodellierung eine Vereinfachung der Modelle erreichen kann.

Zur Illustration des durch diese Vereinfachungen und Abstraktionen erreichbaren Gewinns, wollen wir hier das kleine Beispiel aus Abschnitt 4.2.3 auf Seite 75 einer Java-Realisierung gegenüberstellen. Zunächst die Wiederholung der Darstellung des Beispiels in der Aktionssprache AL++:

```
1   /* KomplexerGrafikElementTyp: */
2   tauscheKreis() {
3       GrafikElementTyp MeinTeil;
4       foreach(MeinTeil in this.teil) {
5           if(MeinTeil.type == KomplexerGrafikElementTyp) {
6               ((KomplexerGrafikElementTyp)MeinTeil).tauscheKreis();
7           } else {
8               SimplerGrafikElementTyp MeinSimplerTeil =
9                 (SimplerGrafikElementTyp)MeinTeil;
10              if(MeinSimplerTeil.name == "Kreis") {
11                  PunktTyp MeinPunktTyp = MeinSimplerTeil.punktTyp;
12                  MeinSimplerTeil.punktTyp -= MeinPunktTyp;
13
14                  SimplerGrafikElementTyp MeineEllipse =
15                    new SimplerGrafikElementTyp("Ellipse");
16                  MeineEllipse.punkttyp += MeinPunktTyp,
17                    ("Assoziation", 1, 3, null, null);
18
19                  foreach(KomplexerGrafikElementTyp MeinGanzes
20                    in MeinSimplerTeil.ganzes) {
21                      MeinGanzes.teil -= MeinSimplerTeil;
22                      MeinGanzes.teil += MeineEllipse,
23                        ("Aggregation", 1, 1, null, null);
24                  }
25              }
```

```
26            }
27        }
28    }
```

Jetzt wollen wir den obigen Algorithmus in der Java-Implementierung angeben. Dabei verwenden wir die oben eingeführten Abbildungen auf Java-Code und verwenden wegen der fehlenden Multiebenenunterstützung das Metamodell aus Abb. 5-14.

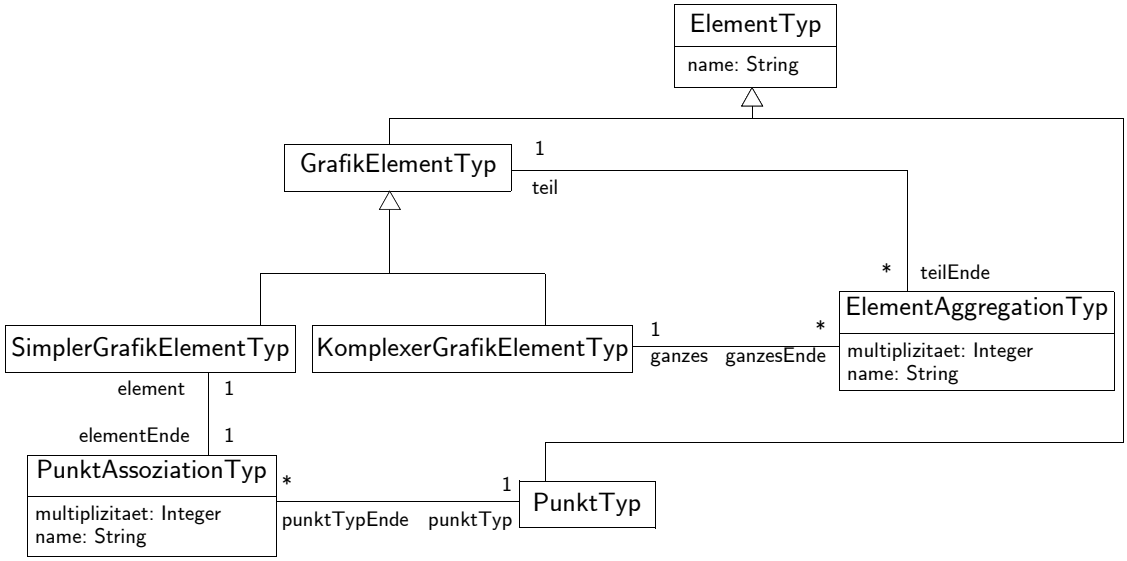


Abbildung 5-14. Metamodell für Grafikbeispiel ohne tiefe Instanziierung (relevanter Ausschnitt von Abb. 3-7 auf Seite 44)

Der vollständige Java-Code der Klasse KomplexerGrafikElementTyp ist damit:

```
1   public class KomplexerGrafikElementTyp {
2       public void tauscheKreis() {
3           Iterator it = this.getGanzesEnde();
4           Set tempSet = new HashSet();
5           while(it.hasNext()) {
6               tempSet.add(it.next());
7           }
8           ElementAggregationTyp teilAggregation;
9           GrafikElementTyp MeinTeil;
10          it = tempSet.iterator();
11          while(it.hasNext()) {
12              teilAggregation = (ElementAggregationTyp)it.next();
13              MeinTeil = teilAggregation.getTeil();
14              if(MeinTeil.getClass()
15                == KomplexerGrafikElementTyp.class) {
16                  ((KomplexerGrafikElementTyp)MeinTeil).tauscheKreis();
17              } else {
```

```
18                  SimplerGrafikElementTyp MeinSimplerTeil =
19                    (SimplerGrafikElementTyp)MeinTeil;
20                  if(MeinSimplerTeil.getName().equals("Kreis")) {
21                      PunktAssoziationTyp altePunktAssoziation =
22                        MeinSimplerTeil.getElementEnde();
23                      PunktTyp MeinPunktTyp =
24                        altePunktAssoziation.getPunktTyp();
25                      MeinSimplerTeil.setElementEnde(null);
26                      MeinPunktTyp.
27                        removePunktTypEnde(altePunktAssoziation);
28
29                      SimplerGrafikElementTyp MeineEllipse =
30                        new SimplerGrafikElementTyp();
31                      MeineEllipse.setName("Ellipse");
32                      PunktAssoziationTyp neuePunktAssoziation = new
33                        PunktAssoziationTyp();
34                      neuePunktAssoziation.setName("Assoziation");
35                      neuePunktAssoziation.setMultiplizitaet(3);
36                      MeineEllipse.
37                        setElementEnde(neuePunktAssoziation);
38                      MeinPunktTyp.
39                        addPunktTypEnde(neuePunktAssoziation);
40
41                      Iterator it2 = MeinSimplerTeil.getTeilEnde();
42                      Set tempSet2 = new HashSet();
43                      while(it2.hasNext()) {
44                          tempSet2.add(it2.next());
45                      }
46                      it2 = tempSet2.iterator();
47                      KomplexerGrafikElementTyp MeinGanzes;
48                      ElementAggregationTyp ganzesAggregation;
49                      while(it2.hasNext()) {
50                          ganzesAggregation =
51                            (ElementAggregationTyp)it2.next();
52                          MeinGanzes = ganzesAggregation.getGanzes();
53                          MeinGanzes.
54                            removeGanzesEnde(ganzesAggregation);
55                          MeinSimplerTeil.
56                                removeTeilEnde(ganzesAggregation);
57                          ElementAggregationTyp neueAggregation = new
58                            ElementAggregationTyp();
59                          neueAggregation.setName("Aggregation");
60                          neueAggregation.setMultiplizitaet(1);
```

```
61                              MeinGanzes.addGanzesEnde(neueAggregation);
62                              MeineEllipse.addTeilEnde(neueAggregation);
63                          }
64                      }
65                  }
66              }
67          }
68      }
```

Alleine der Umfang des Java-Codes (28 gegenüber 68 Zeilen) zeigt deutlich den Abstraktionsgewinn, den man durch Einsatz der Aktionssprache und die verbesserte Multiebenenmodellierung verzeichnen kann.

Neben den oben erläuterten Abbildungen wollen wir insbesondere auf die Zeilen 4–7 und Zeilen 41–45 hinweisen. Die Erzeugung der Mengen-Datenstruktur (`Set`) ist an diesen Stellen deshalb notwendig, da während der Iterator über die Link-Enden läuft, eine Modifikation dieser Enden (in Zeile 53 respektive 61 und Zeile 55 respektive 62) vorgenommen wird. Dies führt in Java unweigerlich zu einem „Concurrent Modification Error“ (siehe auch weiter oben, wo die allgemeine Abbildung des `foreach`-Konstrukts erläutert wurde).

Im obigen Java-Code erkennt man desweiteren die aufwändigen Schritte zur Erzeugung und Entfernung von Links. Beispielhaft seien hier die Zeilen 32 bis 38 genannt. Zunächst muss man eine Instanz des Relationstyps anlegen (mit `new PunktAssoziationTyp()`) und die gewünschten Eigenschaften (mit `setName()` und mit `setMultiplizitaet()`) angeben. Danach müssen Links auf die an der Relationsinstanz teilnehmenden Instanzen angelegt werden (`setElementEnde()` und `addPunktTypEnde()`). Es werden in dieser „traditionellen“ Implementierung also fünf Zeilen Code benötigt, wo bei unserer Aktionssprache eine Zeile genügt.

5.2.3 Aufteilung von Aktivitäten auf Werkzeuge

Nachdem wir nun die Realisierung konkreter Aktionen in Java präsentiert haben, wollen wir in diesem Abschnitt eine grobere Sichtweise einnehmen und zeigen, wie man die Verantwortlichkeiten auf die eigentlichen Automatisierungswerkzeuge verteilt.

Die Motivation für mehrere, kleinere Werkzeuge, die spezifische Aktivitäten automatisieren gegenüber einem einzigen aber mächtigen Werkzeug, ist vielfältig. Zunächst spricht eine sehr stark entkoppelte Werkzeugentwicklung für diese Aufteilung (jeder Entwickler kann *ein* Werkzeug eigenverantwortlich bearbeiten). Desweiteren ist ein sauberes „Separation of Concerns“ möglich, da man exakt das Werkzeug und daher den Code kennt, der für eine entsprechende Transformation zuständig ist. Schließlich bietet sich durch Vorhandensein einzelner kleiner Werk-

zeuge eine sehr einfache Zusammensetzung und Hintereinanderausführung automatisierter Aktivitäten an. So kann man bereits mit einfachen Skripten (z.B. unter UNIX) komplexere Prozessabläufe realisieren. In Abschnitt 9.3.3 auf Seite 285 werden wir sogar einen sehr komplexen Ablauf solcher Aktivitäten, der durch eben ein solches Skripting möglich ist, vorstellen.

Da alle diese Werkzeuge als Basis die Java-Klassen, die aus dem jeweiligen Produktmodell generiert wurden, besitzen, legt dies eine Aufteilung in eine gemeinsame Code-Basis und den eigentlichen Werkzeug-Code nahe. Die *Basis-Klassen* würden dann sowohl die generierten Felder als auch Zugriffs- und Mutatormethoden beinhalten, wohingegen die *Werkzeug-Klassen* die eigentlichen Transformationsmethoden definieren.

Die typische OO-Vorgehensweise an dieser Stelle wäre, die Werkzeug-Klassen von den Basis-Klassen abzuleiten (Vererbung). Angenommen, wir hätten ein Transformationswerkzeug für den im vorhergehenden Abschnitt vorgestellten Algorithmus zur Ersetzung von Kreisen, dann böte sich an, die speziellere Klasse `WerkzeugKomplexerGrafikElementTyp` von `KomplexerGrafikElementTyp` abzuleiten und die Methode `tauscheKreis()` in dieser spezielleren Klasse zu definieren.

Diese an sich elegante Vorgehensweise stößt leider wegen der in Java nicht möglichen Mehrfachvererbung an ihre Grenzen. Nehmen wir z.B. an, wir wollen zusätzlich eine Methode für die Klasse `GrafikElementTyp` definieren, dann müssten wir die speziellere Klasse `WerkzeugGrafikElementTyp` davon ableiten. Desweiteren wollen wir, dass `WerkzeugKomplexerGrafikElementTyp` natürlich eine Spezialisierung von `WerkzeugGrafikElementTyp` ist, damit wir `WerkzeugKomplexerGrafikElementTyp` auch an den Stellen verwenden dürfen wo `WerkzeugGrafikElementTyp` gefordert ist. Damit ergäbe sich der in Abb. 5-15 gezeigte Vererbungsgraph, welcher so nicht in Java zu realisieren ist.

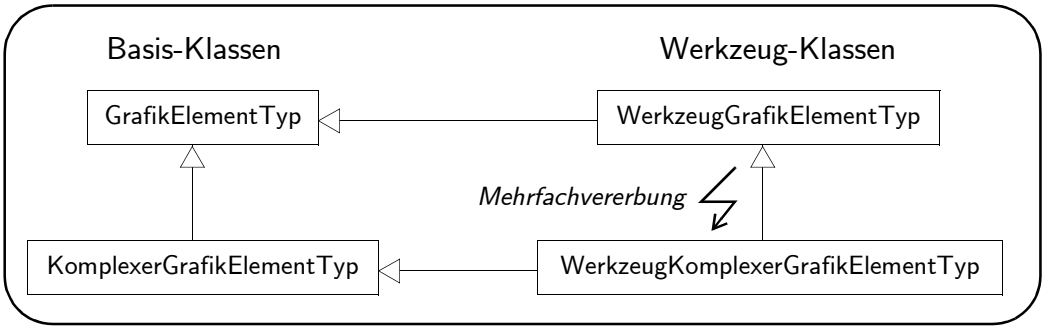


Abbildung 5-15. Mehrfachvererbung bei Trennung in Basis- und Werkzeug-Klassen

Als einfache Lösung gingen wir an dieser Stelle den Weg die Vererbung durch eine *Verschmelzung* von Basis- und Werkzeugklassen zu ersetzen. Genauer gesagt bedeutet dies, dass die in den Werkzeugklassen definierten Methoden in den Klassenrumpf der Basis-Klassen eingefügt werden und das Ergebnis die entsprechend er-

weiterten Klassen sind. In Abb. 5-16 ist dieser Schritt an unserem obigen Beispiel veranschaulicht. Anders als in der ersten Lösung besteht der Werkzeug-Code nur aus den verschmolzenen Klassen (durch das gerundete Rechteck gekennzeichnet).

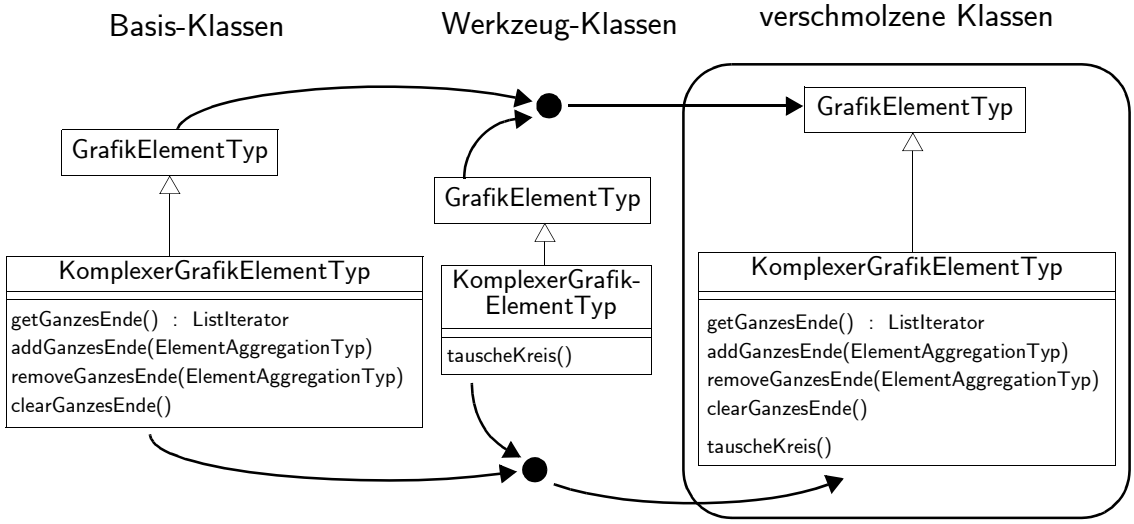


Abbildung 5-16. Lösung des Vererbungsproblems durch Verschmelzung

5.2.4 Austauschformat für Modellinformation

Zum Abschluss dieses Abschnittes zur technischen Realisierung wollen wir auf das bereits angesprochene Austauschformat eingehen. Dieses führten wir ein, um Ketten von Werkzeugaufrufen (d.h. automatisierten Aktivitäten) effizienter ausführen zu können und auch partielle Modelle ohne Informationsverlust zwischen Werkzeugen austauschen zu können.

Das Grundkonzept bei diesem Austauschformat ist, die Instanzeninformation mit möglichst wenig Ballast in eine Textdatei zu schreiben. Anders als z.B. XML, das unter anderem für den Austausch von UML-Modellen Verwendung findet (siehe Abschnitt 5.3.1 auf Seite 118), nutzten wir daher ein schlankeres Format. Desweiteren hat sich das von uns gewählte Format seit langem in unserer Arbeitsgruppe bewährt. Wir wollen dieses Austauschformat *SLANG* (*Simple Data Description Language* [Sch88]) aber nur beispielhaft erläutern, da das Format prinzipiell austauschbar ist (siehe wieder Abschnitt 5.3.1).

Abb. 5-17 zeigt zur Veranschaulichung das bereits in Abschnitt 5.1.1 auf Seite 86 erläuterte beispielhafte Modell und dessen Instanzen. Die SLANG-Datei in Abb. 5-18 beinhaltet dabei die entsprechende Instanzeninformation des M_1-Modells. Wie man erkennt, erhält jede Instanz einen Eintrag, der durch den Typ der Instanz und einem eindeutigen Schlüssel (`KEY`) gekennzeichnet ist. Dieser Schlüssel wird benutzt, um die Links zwischen Instanzen eindeutig zu spezifizieren. So wird z.B. beschrieben, dass die Instanz TemperaturAnforderung mit den Instanzen mit

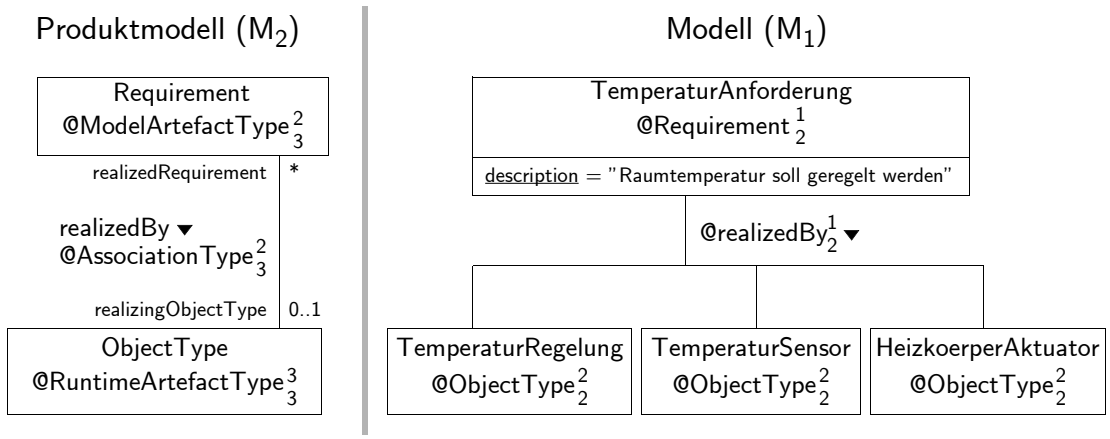


Abbildung 5-17. Darstellung der Modell- und Instanzenebene für ein kleines Beispiel (Wiederholung von Abb. 5-3 auf Seite 89)

den KEYs 2, 3, und 4 über den Rollennamen realizingObjectType verbunden ist. Weiter unten im File werden diese Instanzen dann entsprechend definiert. Man sieht auch, dass die bidirektionalen Links korrekt serialisiert wurden, da auf den beiden Seiten des Links die entsprechende Information festgehalten wurde. Attribute werden ebenso in die Datei geschrieben. So zum Beispiel die Attribute name und description.

Um eine aktuelle Instanziierung in eine SLANG-Datei zu schreiben (*Marshalling*) und auch um aus einer solchen Datei wieder die Instanzen erzeugen zu können (*Unmarshalling*), müssen die entsprechenden Operationen zur Verfügung gestellt werden. Im Folgenden wollen wir eine Realisierung solcher Operationen, die auf dem Produktmetamodell aus Abb. 5-2 auf Seite 88 basiert, vorstellen und damit gleichzeitig die Mächtigkeit unserer Multiebenenbeschreibung und -aktionssprache illustrieren.

SLANG-Datei (M_1)

```
<
 ( Requirement
  <
   ( KEY 1 )
   ( name 'TemperaturAnforderung' )
   ( description 'Raumtemperatur [...]' )
   ( realizingObjectType
    <
     ( KEY 2 ObjectType )
     ( KEY 3 ObjectType )
     ( KEY 4 ObjectType )
    >
   )
  >
 )
                                        ( ObjectType
                                         <
                                          ( KEY 2 )
                                          ( name 'TemperaturRegelung' )
                                          ( realizedRequirement
                                           <
                                            ( KEY 1 Requirement )
                                           >
                                          )
                                         >
                                        )
                                        ...
                                       >
```

Abbildung 5-18. Austauschdatei für das Beispiel aus Abb. 5-17

Eine Voraussetzung für das korrekte Arbeiten der folgenden Operationen ist dabei, dass alle in eine SLANG-Datei zu „marshallenden“ Instanzen ausgehend von einer „Wurzel“-Instanz direkt oder indirekt miteinander verbunden sind. Dies ist notwendig, da wir einen einfachen Graphtraversierungsalgorithmus zum Aufsammeln aller Instanzen einsetzen. Im obigen Beispiel wäre Temperaturanforderung eine solche Wurzel. Für unsere Anwendungen stellt diese Anforderung einer gemeinsamen „Wurzel“ keine Einschränkung dar, weil alle atomaren Artefakte letztlich von einem komplexeren Artefakt aggregiert werden. Alle diese komplexeren Artefakte werden wiederum zu einem einzelnen komplexeren Artefakt zusammengefasst. In Abschnitt 6.2 auf Seite 148 werden wir diese Artefakte für unsere Beispielmethode vorstellen.

Alle für ein Marshalling benötigten Operationen können innerhalb eines beliebigen Objekttyps definiert werden, da wir die jeweils zu bearbeitende Instanz als Aufrufparameter an die entsprechenden Operationen weitergeben. Beginnen wollen wir mit der Beschreibung der `marshal()`-Operation:

```
1   private Set objectsToBeMarshalled = new HashSet();
2   private Set marshalledObjects = new HashSet();
3   private Writer out;
4   public void marshal(Writer pOut, ArtefactType.anyinstance root) {
5       out = pOut;
6       out.println("<");
7       objectsToBeMarshalled.add(root);
8       ArtefactType.anyinstance object;
9       while(!objectsToBeMarshalled.isEmpty()) {
10          foreach(object in objectsToBeMarshalled) {
11              marshalObject(object);
12              break;
13          }
14          marshalledObjects.add(object);
15          objectsToBeMarshalled.remove(object);
16      }
17      out.println(">");
18  }
```

Zunächst werden zwei Mengen definiert, welche die noch in die Austauschdatei zu schreibenden Instanzen und die bereits „gemarshallten“ Objekte beinhalten. Die erste Instanz, deren Daten geschrieben werden, ist die „Wurzel“-Instanz `root`, weshalb sie zu der Menge `objectsToBeMarshalled` hinzugefügt wird. Mit dem Aufruf der `marshalObject()`-Operation beginnt das eigentliche Marshalling:

```
19  private void marshalObject(ArtefactType.anyinstance object) {
20      out.println(" ( "+object.type.name);
21      out.println("  <");
```

```
22      out.println("   ( KEY "+object.key+" )");
23      ArtefactType currentType = object.type;
24      out.println("   ( name '"+object.name+"' )");
25      out.println("   ( description '"+object.description+"' )");
26      while(currentType != null) {
27          marshalAttributes(object, currentType);
28          marshalLinks(object, currentType);
29          currentType = currentType.supertype;
30      }
31      out.println("  >");
32      out.println(" )");
33  }
```

Erst wird der Name des Typs des `objects` und der eindeutige `KEY` der Instanz festgestellt und in die Ausgabedatei (`out`) geschrieben. Als Nächstes werden die impliziten Attribute name und description behandelt. Danach wird die Generalisierungshierarchie bis zur Wurzel verfolgt und auf jeder Ebene die expliziten Attribute und Relationen des entsprechenden Typs serialisiert. Dies ist notwendig, da auf der Ebene des einzelnen Typs nur dessen „eigene“ Attribute und Relationen sichtbar sind.

```
34  private void marshalAttributes(ArtefactType.anyinstance object,
35   ArtefactType currentType) {
36      foreach(String attrName in currentType.AttributeType) {
37          DataType.anyinstance attrValue = object.#attrName;
38          out.println("   ( "+attrName+" '"+attrValue+"' )");
39      }
40  }
```

Zur Serialisierung der jeweiligen Attributbelegungen besorgt man sich zunächst die Menge der Namen der Instanzen des Attributtyps AttributeType. Für jedes Attribut wird dann dessen Belegung für die aktuelle Instanz `object` mit Hilfe des generischen Operators „`#`“ bestimmt und zusammen mit dem Attributnamen in die Datei geschrieben. Vereinfacht wird diese Stelle im Code, da nur Instanzen von Datatype als Datentypen für Attribute zulässig sind (es genügt daher der Ausdruck `DataType.anyinstance`). Ließe man auch allgemeine Artefakte als Attributbelegungen zu, müsste man hier wiederum rekursiv absteigen.

```
41  private void marshalLinks(ArtefactType.anyinstance object,
42   ArtefactType currentType) {
43      foreach(String roleName in currentType.AssociationType) {
44          Multiplicity mult;
45          (null, null, mult, null, null) = (currentType.AssociationType,
46           roleName);
47          out.println("   ( "+roleName);
```

```
48          out.println("    <");
49          if(mult.upper > 1) {
50              foreach(ArtefactType.anyinstance linkedObject in
51               object.#roleName) {
52                  out.println("     ( KEY "+linkedObject.key+" "+
53                   linkedObject.type.name+" )");
54                  if(!marshalledObjects.contains(linkedObject))
55                      objectsToBeMarshalled.add(linkedObject);
56              }
57          } else {
58              ArtefactType.anyinstance linkedObject = object.#roleName;
59              out.println("     ( KEY "+linkedObject.key+" "+
60               linkedObject.type.name+" )");
61              if(!marshalledObjects.contains(linkedObject))
62                  objectsToBeMarshalled.add(linkedObject);
63          }
64          out.println("    >");
65          out.println("   )");
66      }
67  }
```

Das Serialisieren der Links erfolgt ganz analog zu den Attributen. Es ist allerdings etwas aufwändiger, da für jede Assoziation (auf Produktmodellebene) mehrere Links auf Ebene M_1 existieren können. Daher ist z. T. eine geschachtelte Realisierung notwendig. Für jede Assoziation stellt man dazu zunächst Rollenname und Multiplizität fest (Zeilen 43 bis 46). Handelt es sich um eine Assoziation, die an ihrem Ende mehr als eine Instanz erlaubt (obere Multiplizitätsgrenze ist größer eins), dann läuft man über alle Link-Enden und schreibt diese Instanzeninformation in die Datei. Bei einer Multiplizität von max. eins greift man direkt auf die Instanz am Link-Ende zu. Abschließend wird jeweils geprüft, ob man einen Verweis auf eine Instanz gefunden hat, die noch nicht „gemarshalled“ wurde und nimmt diese daher in die Menge der `objectsToBeMarshalled` auf.

Analog verfährt man für Instanzen des Rollentyps AggregationType, welche eine Aggregation von Artefakten beschreiben. Der Übersichtlichkeit halber ist dies in obigem Code nicht gezeigt.

Erwähnt werden sollte an dieser Stelle noch die Realisierung des key-Attributs, welches jeweils einen eindeutigen Schlüssel einer Instanz speichert. In Abb. 5-19 ist diese Realisierung mit Hilfe der Konstrukte der Multiebenenmodellierung gezeigt.

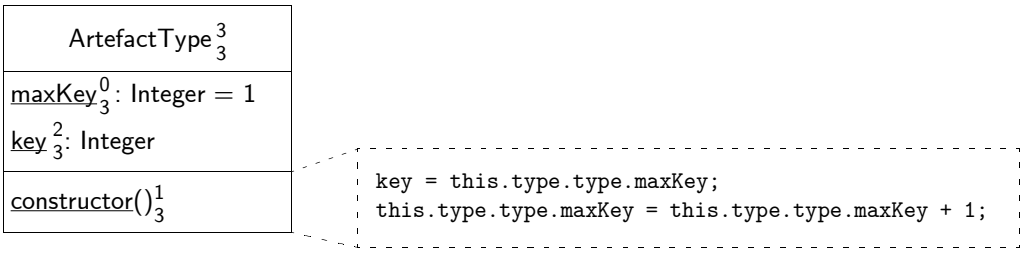


Abbildung 5-19. Berechnung eines eindeutigen Schlüssels für Instanzen

Die Attributinstanz maxKey (Potenz „0“) mit einer Initialbelegung von 1 dient hierbei als Zähler aller Artefaktinstanzen. Das Attribut key wird als Attribut der Potenz „2“ definiert und kann somit für jede M_1-Instanz eine eindeutige Nummer speichern. Für die Zuweisung dieser Nummer definieren wir einen Konstruktor für die benutzerdefinierte Instanziierung der Artefakte. Die Definition des Konstruktors erfolgt hierbei mit der Potenz „1“, wodurch diese Operation nur aus dem Kontext der Instanzen von ArtefactType aufgerufen werden kann. Dies entspricht der üblichen Interpretation eines Konstruktors als „Klassenmethode“ (siehe dazu auch [RJB99, S.245]). Die Aktionen innerhalb des Konstruktors beziehen sich auf die von diesem erzeugte Instanz, weshalb ein direkter Zugriff auf das Attribut key möglich ist und für den Zugriff auf maxKey mittels `this.type.type` zum Objekttyp ArtefactType navigiert werden muss.

Nun wollen wir die `unmarshal()`-Operation als „Umkehrfunktion“ der `marshal()`-Operation vorstellen:

```
1   private MyStreamTokenizer in;
2   public ArtefactType.anyinstance unmarshal(MyStreamTokenizer pIn)
3    throws Exception {
4       in = pIn;
5       ArtefactType.anyinstance root = null;
6       if(!(in.nextToken().equals("<") && in.nextToken().equals("(")))
7           throw new Exception();
8       root = unmarshalObject();
9       String token;
10      while(true) {
11          token = nextToken();
12          if(token.equals(">"))
13              break;
14          else if(token.equals("("))
15              unmarshalObject();
16          else
```

```
17              throw new Exception();
18      }
19      return root;
20  }
```

Zunächst wird die korrekte „Klammerung“ der SLANG-Daten geprüft, und bei einem Fehler eine Exception erzeugt, die der Einfachheit halber an anderer Stelle behandelt wird. Dann wird, solange Daten für Objekte vorhanden sind, diese aus der Datei gelesen und die entsprechenden Instanzen mit der `unmarshalObject()`-Operation erzeugt. Der erste Rückgabewert entspricht dabei der ursprünglich in die Datei geschriebenen „Wurzel“-Instanz `root`.

```
21  private ArtefactType.anyinstance unmarshalObject()
22    throws Exception {
23      String typeName = nextToken();
24      if(!(in.nextToken.equals("<") && in.nextToken().equals("(") &&
25       in.nextToken().equals("KEY")))
26          throw new Exception();
27      String key = in.nextToken();
28      if(!in.nextToken().equals(")"))
29          throw new Exception();
30      ArtefactType.anyinstance object = checkAndCreate(key, typeName);
31      Set allAttributeNames = object.type.AttributeType;
32      Set allAssociationNames = object.type.AssociationType;
33      String token = null;
34      while(true) {
35          token = nextToken();
36          if(token.equals(">"))
37              break;
38          else if(token.equals("("))
39              unmarshalField(object, allAttributeNames,
40               allAssociationNames);
41          else
42              throw new Exception();
43          if(!in.nextToken().equals(")"))
44              throw new Exception();
45      }
46      if(!in.nextToken().equals(")"))
47          throw new Exception();
48      return object;
49  }
```

Für jeden Objekteintrag wird nun zunächst dessen `KEY` ausgelesen und mit Hilfe der `checkAndCreate()`-Operation (s. u.) eine Referenz auf eine Instanz des entspre-

chenden Typs zurückgeliefert. Für diese Instanz (genauer deren Typ) werden dann zunächst Mengen von Attribut- und Assoziationsnamen berechnet, welche später zur Unterscheidung dieser beiden Objekteigenschaften benötigt werden. In einer Schleife werden dann alle Objekteigenschaften mit Hilfe der `unmarshalField()`-Operation aus der Datei gelesen.

```
50  private void unmarshalField(ArtefactType.anyinstance object, Set
51   allAttributeNames, Set allAssociationNames) throws Exception {
52      String fieldName = nextToken();
53      if(fieldName.equals("name")) {
54          String value = convertValue(in.nextToken(), "String");
55          object.name = value;
56      } else if(fieldName.equals("description")) {
57          String value = convertValue(in.nextToken(), "String");
58          object.description = value;
59      } else if(allAttributeNames.contains(fieldName)) {
60          DataType dataType;
61          dataType = (object.type.AttributeType, fieldName);
62          #(dataType.name) value = convertValue(in.nextToken(),
63           dataType.name);
64          object.#fieldName = value;
65      } else if(allAssociationNames.contains(fieldName)) {
66          if(!in.nextToken().equals("<"))
67              throw new Exception();
68          String token;
69          while(true) {
70              token = nextToken();
71              if(token.equals(">"))
72                  break;
73              else if(token.equals("("))
74                  unmarshalLinks(fieldName, object);
75              else
76                  throw new Exception();
77              if(!in.nextToken().equals(")"))
78                  throw new Exception();
79          }
80      } else {
81          throw new Exception();
82      }
83  }
```

Beschreibt `field` ein implizites Attribut, also name oder description, so wird dies zunächst behandelt. Handelt es sich bei „`field`“ um ein explizites Attribut (als Instanz des AttributeType, siehe Zeilen 59–64), dann muss zunächst dessen Datentyp

festgestellt werden. Der Vollständigkeit halber müsste man auch hier durch die gesamte Generalisierungshierarchie laufen (so wie in den Zeilen 23–30 der `marshal()`-Operation). Zur Wahrung der Übersichtlichkeit lassen wir diesen Schritt hier aber aus. Hat man den Datentyp festgestellt, dann kann man eine Referenz auf den Wert des Attributs erzeugen (mit dem generischen „`#`“-Operator) und nach einer entsprechenden Konvertierung des Strings mit Hilfe der `convertValue()`-Operation (hier nicht näher erläutert) dieser Referenz zuweisen. Abschließend wird dieser Wert dann als Attributbelegung verwendet.

Für das Unmarshalling von Links wird analog vorgegangen. Die Operation `unmarshalLinks()` realisiert die notwendigen Aktionen für die einzelnen Assoziationen.

```
84  private void unmarshalLinks(String fieldName,
85   ArtefactType.anyinstance object) throws Exception {
86      if(!in.nextToken().equals("KEY"))
87          throw new Exception();
88      String key = in.nextToken();
89      String typeName = nextToken();
90      if(!in.nextToken().equals(")"))
91          throw new Exception();
92      ArtefactType.anyinstance linkedObject =
93       checkAndCreate(key, typeName);
94      Multiplicity mult;
95      (null, null, mult, null, null) = (object.type.AssociationType,
96       fieldName);
97      if(mult.upper > 1)
98          object.#fieldName += linkedObject;
99      else
100         object.#fieldName = linkedObject;
101 }
```

Wie auch schon beim Marshalling werden für die Links unterschiedliche Aktionen in Abhängigkeit von der Multiplizität der Assoziation notwendig. Hier müssen je nach Multiplizität unterschiedliche Zuweisungsoperatoren verwendet werden. Da die Link-Enden auf eine Objektinstanz verweisen, wird an dieser Stelle wieder die `checkAndCreate()`-Operation notwendig, welche eine Referenz auf eben ein solches Objekt liefert. Diese Operation wird wie folgt realisiert:

```
102 private Map unmarshalledObjects = new Hashtable();
103 private ArtefactType.anyinstance checkAndCreate(String key,
104  String typeName) {
105     ArtefactType.anyinstance object = null;
106     if(unmarshalledObjects.containsKey(key)) {
107         object = unmarshalledObjects.get(key);
```

```
108      } else {
109          object = new #typeName("");
110          unmarshalledObjects.put(key, object);
111      }
112      return object;
113  }
```

Bei der „Berechnung“ einer Objektreferenz ausgehend von einem eindeutigen Schlüssel können zwei Fälle unterschieden werden. Existiert eine Instanz mit entsprechendem Schlüssel noch nicht, dann muss diese entsprechend ihres Typs erzeugt werden (Zeilen 108–111) und in die Tabelle der existierenden Objekte aufgenommen werden. Im anderen Fall genügt es, die Referenz aus der Tabelle zu lesen und zurückzuliefern.

Die Operationen wurden, wie in Abschnitt 5.2.2 erläutert wurde, mit Hilfe des Reflexions- und Introspektionsmechanismus auf Java abgebildet. Dazu waren für die Implementierung der `marshal()`-Methode 91 Zeilen (im Vgl. zu 67 Zeilen) und für die `unmarshal()`-Methode 140 Zeilen (im Vgl. zu 113 Zeilen) notwendig.

5.3 Verwandte Arbeiten

Nachdem wir in den vorangegangenen Abschnitten eine spezielle Technik zur Automatisierung kennenlernten, soll ein Vergleich mit verwandten Arbeiten dieses Kapitel abschließen. Neben bekannten Meta-Metamodellen werden wir auch damit zusammenhängende Techniken zur Automatisierung diskutieren und an unsere Arbeiten angrenzende Bereiche vorstellen.

5.3.1 Meta-Metamodelle

Im sich anschließenden Kapitel werden wir ein konkretes Metamodell für die Entwicklungsmethode PROBAnD vorstellen, welche eine Instanziierung der Meta-Metamodelle dieses Kapitels (Abschnitt 5.1.1 auf Seite 86) darstellt. Hier soll daher zunächst eine Übersicht über existierende und weit verbreitete Meta-Metamodelle gegeben werden.

Meta Object Facility (MOF)

Mit der *Meta Object Facility* (*MOF* [OMG02a]) stellt die OMG (Object Management Group) ein Meta-Metamodell (Ebene M_3) für die eigenen Modellierungssprachen wie UML oder CORBA IDL zur Verfügung [BHK04, S.8ff.].

Die wichtigsten in MOF definierten Objekttypen (Meta-Metatypen) sind dabei MetaClass und MetaAssociation [Fra99]. Für die Definition von Metamodellen (Ebene M_2) wird dabei die traditionelle Metamodellierung eingesetzt (siehe Abschnitt

3.2.3 auf Seite 51), was insbesondere bedeutet, dass eine Assoziation auf der M_2-Ebene als Instanz des M_3-Typs MetaClass definiert wird (siehe auch Abschnitt 3.2.1 auf Seite 40).

Neben der Definition von Metamodellen beinhaltet der MOF-Standard auch verschiedene sog. *Technologieabbildungen*. Am weitesten verbreitet ist dabei das schon öfter zitierte XMI (XML Metadata Interchange, siehe auch Abschnitt 5.2.4 auf Seite 109). XMI definiert, wie Instanzen von MOF-Elementen in XML-Dokumenten beschrieben werden [BHK04, S. 81]. Diese Abbildung soll die Grundlage für den Austausch von UML-Modellen zwischen Werkzeugen unterschiedlicher Hersteller bilden. In der Praxis kämpft man allerdings noch mit nicht kompatiblen Formaten und insbesondere verschiedenen Versionen der XMI-Dateien. Desweiteren können bisher keine Diagramme mit XMI ausgetauscht werden.

Eine weitere Abbildung liefert das *Java Metadata Interface* (*JMI*, siehe [Dir02]), welches die standardisierte Erzeugung von Java-Interfaces definiert, mit Hilfe derer auf die zugehörigen Instanzen eines Repositorys zugegriffen werden können. Die Methoden für den Zugriff auf die Repository-Information sind dabei sehr ähnlich zu denen, die wir in unserem Ansatz verwenden (siehe dazu Abschnitt 5.2.1 auf Seite 97). Unsere bisher vorgestellte Technik zur Realisierung der Automatisierungswerkzeuge ließe sich daher auch mit Hilfe von JMI realisieren. Da eine JMI-Implementierung zu Beginn dieser Arbeiten nicht zur Verfügung stand, gingen wir allerdings den in Abschnitt 5.2 auf Seite 96 beschriebenen werkzeugspezifischen Weg.

Der Vorteil von JMI liegt ganz klar auf einer standardisierten und damit werkzeugunabhängigen Abbildung. Desweiteren existiert auf der Basis des MOF Reflective Package [Dir02, S. 77ff.] auch die Möglichkeit auf Einträge des Repositorys zuzugreifen, deren Typen zur Kompilierzeit noch nicht feststehen. Eine Erzeugung neuer Meta-Metainstanzen ist damit allerdings nicht möglich.

Schließlich wird auch eine Technologieabbildung auf die OMG-Middleware *CORBA* (*Common Object Request Broker Architecture* [OMG02]) definiert, welche den Zugriff auf Repositorys auf der Basis von verteilten Objekten erlaubt.

CASE Data Interchange Format (CDIF)

Das Hauptziel des von der *EIA* (*Electronic Industries Alliance*) spezifizierten *CDIF* (*CASE Data Interchange Format* [EIA04]) ist es, Daten zwischen unterschiedlichen Modellierungswerkzeugen und Repositorys austauschen zu können. Zu diesem Zweck definiert CDIF ein integriertes Metamodell und Abbildungen auf Dateiformate oder verteilte Objekttechnologien (z. B. CORBA). Dabei beinhaltet CDIF sowohl die abstrakte als auch die konkrete Syntax von Modellen, wobei auch mehrere Darstellungen für *ein* abstraktes Konzept möglich sind. Die beiden wichtigsten Meta-Metatypen bei CDIF sind MetaEntity und MetaRelationship. (siehe dazu auch

die Arbeit von Flatscher [Fla02], welche eine Einführung in die grundlegenden CDIF-Konzepte bietet).

An Hand dieser Beschreibung wird bereits deutlich, dass CDIF Pate für viele der in MOF eingeflossenen Konzepte stand oder zumindest einige der Konzepte parallel zueinander entwickelt wurden.

5.3.2 Metamodellbasierte Transformationsansätze

In den folgenden beiden Abschnitten werden existierende Transformationsansätze vorgestellt und mit unserem Ansatz verglichen.

Model Driven Architecture (MDA)

Basierend auf den Erfolgen der Modellierungssprache UML wird von der OMG derzeit die *Model Driven Architecture* (*MDA*, siehe [RSN03], [KWB03] und [MiM03]) propagiert, bei welcher die primären Artefakte Modelle und nicht mehr Code-Fragmente sind.

Das Hauptziel ist es, mit Hilfe solcher Modelle Software plattformunabhängig zu spezifizieren und eine Abbildung auf konkrete Middleware- und Hardware-Plattformen vorzunehmen (siehe Abbildung 5-20). Als Technologien werden die OMG eigenen Standards wie MOF, XMI und UML eingesetzt und um spezifische Bedürfnisse erweitert, z.B. um eine Technik zur Transformation von Modellen (siehe „Request for Proposals“ zu „Queries/Views/Transformations“ [ACR03]).

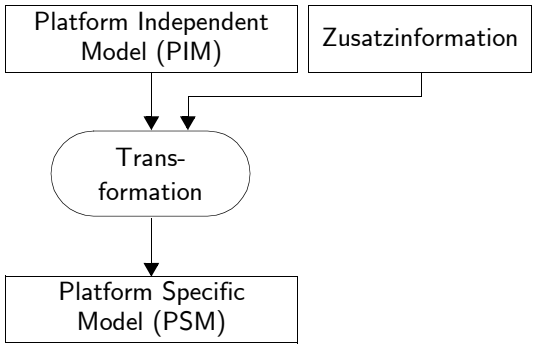


Abbildung 5-20. MDA-Transformation [MiM03, S.2-7]

In der MDA werden zur Erreichung dieses Ziels zwei wichtige Klassen von Modellen definiert, das *PIM* (*Platform Independent Model*) und das *PSM* (*Platform Specific Model*). Im PIM werden die Struktur und das Verhalten implementierungsunabhängig spezifiziert (z.B. mit Techniken wie der UML Action Semantics, siehe auch Abschnitt 4.2.2 auf Seite 65). Das PSM ergänzt die Informationen im PIM um plattformspezifische Aspekte, s.d. daraus ein lauffähiges System erzeugt werden kann [BHK04, S.283f.]. In diesem Sinne kann man sich ein PSM auch als eine Art Entwurfsmodell vorstellen, wohingegen das PIM als Analysemodell interpretiert werden könnte.

Die Anforderungen an das System können in einem weiteren Typ von Modell, dem *CIM* (*Computing Independent Model*) beschrieben werden, welches Use-Case-, Interaktions- und Aktivitätsdiagramme beinhaltet [BHK04, S. 281ff.].

Die Erkenntnis, dass die Abstraktion in der Software-Entwicklung weg vom Code zu Modellen erfolgen muss, ist sicherlich ein begrüßenswerter Trend in der Industrie. Die Forschung beschäftigt sich allerdings schon viel länger mit solchen Ansätzen. Als Beispiel sei hier das in unserer Arbeitsgruppe entwickelte *MOOSE* (*Model-based Object-Oriented Software Generation Environment* [ARS97]) genannt, das bereits Mitte der 90er Jahre automatisch C++-Code aus EER-Diagrammen erzeugen konnte (inklusive Zugriffs- und Mutatormethoden und Methoden für das Erzeugung des SLANG-Austauschformats). Das kommerzielle Werkzeug Statemate erlaubte schon 1987 die Generierung von Code aus Modellen, die neben der Struktur des Systems auch dessen Verhalten (mit einer Variante der Zustandsautomaten) beschreiben [Har01].

Hauptaufgabe bei der Arbeit mit MDA ist die Definition der Abbildungen zwischen PIM und PSM. Viele der existierenden Werkzeuge zur Code-Generierung, welche diesen Schritt automatisieren, werden daher momentan unter dem Etikett „MDA“ vermarktet, auch wenn nicht in allen Fällen der Weg über das PSM beschritten wird, sondern eine direkte Abbildung auf Code erfolgt (so z. B. bei AndroMDA [Boh04], ArchitectureWare [Völ04] oder Rhapsody [GHP02]). Eine nennenswerte Ausnahme ist das Produkt ArcStyler der Interactive Objects Software GmbH [Hub02], da hierbei der vollständige Entwicklungsprozess durch generative Schritte (ausgehend von einem initialen Geschäftsprozessmodell) unterstützt wird.

Um von einem abstrakten Modell zu ausführbaren Artefakten zu gelangen, muss die „semantische Lücke“ zur Implementierungssprache geschlossen werden. Eine automatische Abbildung von PIM zu PSM kann allerdings immer nur semantikerhaltend erfolgen, weil ansonsten ein kreativer Schritt durch ein Werkzeug nötig wäre. Daher muss das Platform Independent Model entweder um entsprechende Informationen angereichert werden, womit sich die Modellierung allerdings fast schon zur grafischen Programmierung [Ste04] entwickelt, oder muss der generierte Code manuell ergänzt werden, was in vielen der Werkzeugen durch spezielle „geschützte“ Code-Regionen ermöglicht wird, die auch bei einer Re-Generierung erhalten bleiben.

Im Gegensatz zur MDA sehen wir die Hauptanwendung und den wesentlichen Beitrag unserer Arbeit in der Automatisierung der Modell*evolution* und -*analyse* und nicht so sehr der Modell*implementierung* (siehe auch Ende von Abschnitt 2.2.2 auf Seite 19, wo wir diesen Schwerpunkt bereits erwähnten).

Andere Transformationsansätze

Neben der MDA gibt es eine Vielzahl anderer Transformationsansätze. Der Großteil dieser Ansätze basiert auf Graphtransformationen, so z. B. jene, die von Sprinkle et

al. [SAL02], Varro et al. [VGP01], Appukuttan et al. [ACR03] oder Hausmann et al. [HHS02] vorgeschlagen werden. In Abschnitt 4.2.1 auf Seite 62 hatten wir bereits ausreichend dargelegt, weshalb wir diesen Ansatz zur Modelltransformation in unserer Arbeit nicht verfolgen werden.

Außer diesen graphbasierten Ansätzen existieren allerdings auch Umgebungen, die eine operationale Spezifikation der Transformation erlauben. Zu diesen gehört z.B. eine von Sunye et al. propagierte Lösung [SGJ02], welche auf der Verwendung der UML Action Semantics fußt und daher zunächst nur für UML-Modelltransformationen eingesetzt wird. Weitere Techniken zur operationalen Transformation werden im folgenden Abschnitt unter dem Thema MetaCASE-Werkzeuge aufgeführt.

Manche Umgebungen verwenden gar ganz andere Ansätze, wie z.B. *UMLAUT* [HJG99], wo eine Komposition von Funktionen zur Beschreibung von Transformationen verwendet wird, oder *Domain Mapping Specifications* [Mil02], die eine grafische Beschreibung der Transformation (angereichert mit operationalen Konzepten wie Schleifen) ermöglicht. Diese Ansätze bringen in unseren Augen allerdings dieselben Nachteile wie proprietäre Sprachen zur operationalen Transformation mit sich (so kann man z.B. nicht auf existierende Bibliotheken für Standardalgorithmen zugreifen und muss zunächst die neue Sprache lernen).

Allerdings kann eine grafische Transformationssprache (wie von Milicev [Mil02] und auch von Appukuttan et al. [ACR03] vorgeschlagen) sicherlich die Erstellung einfacher Transformations- und damit Automatisierungsaufgaben erleichtern, da ein Entwickler dann zunächst ohne Programmierkenntnisse auskommen könnte. Wie weit die grafische Beschreibung der Transformationen skaliert ist allerdings fraglich.

5.3.3 (Meta-)CASE Werkzeuge

Neben der Existenz von sinnvollen Metamodellen und Transformationsansätzen sind auch Werkzeuge, welche solche Techniken umsetzen, nötig, um eine Automatisierung effizient realisieren zu können. In diesem Abschnitt wollen wir daher eine Übersicht über das weite Feld existierender CASE- und MetaCASE-Werkzeuge geben (*CASE* steht für *Computer Aided Software Engineering*).

Wir beginnen mit einer Klassifikation der Werkzeuge, wozu wir in Tabelle 5-1 die verschiedenen Werkzeugklassen bezüglich ihrer Flexibilität einordnen.

In die Klasse der CASE-Werkzeuge fallen alle heute schon quasi als Standard anzusehenden Werkzeuge, die ausgehend von einem festgelegten Metamodell, i.d.R. UML, eine Abbildung der Eingabemodelle in lauffähigen Code vornehmen. Beispiele sind das in dieser Arbeit verwendete Rhapsody (von i-Logix), aber auch Rational Rose (von IBM), Poseidon (von gentleware) und Tau Generation 2 (von Telelogic).

Der Vollständigkeit halber müsste man in diese Aufzählung auch Werkzeuge wie Hochsprachen-Compiler (z.B. C++) aufnehmen, da diese ausgehend von dem Metamodell der Hochsprache eine Ausgabe in Maschinensprache liefern (typischerweise wäre das Metamodell dieser Werkzeuge aber eher eine Grammatik der Hochsprache, siehe dazu auch Abschnitt 2.1.2 auf Seite 13).

Tabelle 5-1. Klassifikation der Werkzeuge (in Anlehnung an [Mil02])

Klasse	**Metamodell**	**Ausgabe**
CASE: einfaches, nicht-anpassbares Modellierungswerkzeug	festgelegt	festgelegt
A-CASE: anpassbares Modellierungswerkzeug	festgelegt	anpassbar
MetaCASE: Metamodellierungswerkzeug mit festgelegter Abbildung (z.B. Standard-Berichtsgeneratoren)	anpassbar	festgelegt
A-MetaCASE: anpassbares Metamodellierungswerkzeug	anpassbar	anpassbar

Viele der oben erwähnten CASE-Werkzeuge bieten auch eine gewisse Anpassbarkeit der Ausgaben (sind also als A-CASE klassifizierbar). So kann man in Rhapsody z.B. die Implementierung von Relationen verändern und an Stelle einer Liste auch die Verwendung anderer Datenstrukturen parametrisieren. Neben der reinen Code-Generierung bieten solche Werkzeuge auch häufig sog. Report-Generatoren an, welche das Modell in Dokumente überführen, die man zu Dokumentationszwecken einsetzen kann. Auch die MDA-Werkzeuge, welche im letzten Abschnitt aufgeführt wurden, fallen üblicherweise in diese Kategorie, da die Sprache zur Spezifikation eines PIMs i.d.R. UML ist.

Der Nachteil aller dieser CASE-Tools ist, dass das Metamodell fest codiert ist und man diese daher nur für die vom Werkzeug unterstützte Modellierungssprache einsetzen kann. Manche der UML-Werkzeuge sehen daher eine „Erweiterung“ mit Hilfe von Stereotypes vor (insbesondere trifft dies auf die MDA-Werkzeuge zu). Der Stereotype-Mechanismus erlaubt allerdings immer nur eine Ergänzung des bestehenden Metamodells und nicht die benutzerdefinierte Änderung existierender Metatypen (siehe dazu auch [AKH03]).

Werkzeuge, die eine Änderung oder Anpassung des Metamodells bei fest kodierter Ausgabe erlauben sind nicht häufig anzutreffen, da mit einer fest kodierten Ausgabe die Mächtigkeit der Metamodellierung kaum ausgenutzt werden kann. Meist bezieht sich diese Ausgabe daher auf allgemein gehaltene Berichtsgeneratoren oder entsprechende Austauschformate. Eine solches MetaCASE-Tool dient damit also eher der Eingabe und manuellen Bearbeitung von Modellen.

Die für diese Arbeit interessanteste Klasse von Werkzeugen ist die der „echten“ MetaCASE-Tools, die sowohl eine benutzerdefinierte Anpassung des Metamodells als auch eine Spezifikation der Abbildungen und Ausgaben erlauben (A-MetaCASE-Werkzeug). Solche Werkzeuge besitzen daher ein fest kodiertes Meta-Metamodell (M_3-Ebene), welches den in Abschnitt 3.2 auf Seite 38 erwähnten minimalen Satz an Modellierungselementen enthält. Damit lassen sich dann eigene Metamodelle als Instanzen dieser Modellierungselemente definieren (siehe Abb. 5-21).

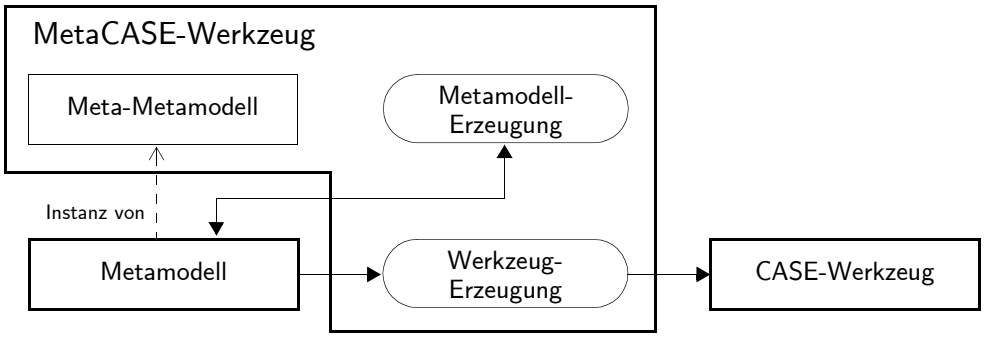


Abbildung 5-21. Einsatz eines MetaCASE-Werkzeugs

Ausgehend von einem benutzerdefinierten Metamodell werden die zur Eingabe und Bearbeitung der Modelle benötigten Editoren und Werkzeuge automatisch generiert. Ein MetaCASE-Werkzeug ist damit also ein Metawerkzeug, das der Erzeugung von CASE-Werkzeugen dient [EnK00].

Vertreter dieser Klasse von Werkzeugen sind:

- *DOME* (*Domain Modeling Environment* [EnK00])
  `http://www.htc.honeywell.com/dome/`
- *GME* (*The Generic Modeling Environment* [LMB01])
  `http://www.isis.vanderbilt.edu/projects/gme/`
- *MetaEdit+* und *MethodWorkbench* [KeT00]
  `http://www.metacase.com/`
- *EMF* (*Eclipse Modeling Framework*) [MDG04]
  `http://www.eclipse.org/emf/`
- *KOGGE* (*KOblenzer Generator für Graphische Entwurfsumgebungen*)
  `http://www.uni-koblenz.de/~ist/kogge.html`

Bis auf die jeweils verwendete Technik ähneln sich viele Konzepte der Werkzeuge. So erzeugt z.B. jedes der obigen MetaCASE-Werkzeuge einen grafischen Modelleditor (inkl. Hierarchiekonzepten) aus den Metamodellen. Große Unterschiede bestehen vor allem bzgl. der Transformation der Modelle, welches eine für unsere Ansätze relevante Basistechnologie darstellt.

Wir wollen hier daher stellvertretend drei Werkzeuge auswählen und genauer vergleichen. In Tabelle 5-2 sind dazu die in unseren Augen am weitest fortgeschrittenen Werkzeuge gegenübergestellt.

Tabelle 5-2. Vergleich einiger anpassbarer MetaCASE-Werkzeuge (g: grafisch, t: textuell)

Eigenschaft	**Werkzeug**		
	DOME	**GME**	**MetaEdit+**
Notation (M_3-Modell)	Variante von Coad-Yourdon OOA: Nodes und Connections (g)	Variante der UML-Klassendiagramme: Models, Atoms und Connections (g)	EER-Dialekt: Objects, Relations, Roles und Ports (t)
Transformationssprache	Data-Flow-Sprache (g) und Scheme-Dialekt (t)	keine	keine
Dokumentgeneration	mittels Transformationssprache oder template-basiert	musterbasiert	eigene Report-Generierungssprache
Zugriff auf Repository	interne Erstellung eines Data-Dictionarys möglich	externer Zugriff über COM (Komponentenmodell)	externer Zugriff über SOAP (Web-Services)
Austauschformat	bel. Export (Dokumentgenerierung)	Ein- und Ausgabe von XML	vordefinierter Export nach XML

Die ersten beiden Vertreter, DOME und GME, sind Ergebnisse von Forschungsabteilungen bzw. Universitäten, wohingegen MetaEdit+ ein kommerzielles Werkzeug darstellt. Dennoch bieten die nicht-kommerziellen Vertreter einen erheblichen Funktionsumfang, der sogar über den von MetaEdit+ hinausgeht.

DOME glänzt durch zwei Sprachen zur Modelltransformation (eine grafische und eine textuelle). Da es sich bei den Modellen allerdings immer um Graphen handelt, ist eine Modifikation nicht ganz einfach, da man stets die konkrete Syntax beachten muss (also z.B. explizite Koordinaten für die Platzierung neuer Komponenten angeben muss). Auch der verwendete Scheme-Dialekt [KCR98], der in seiner Syntax große Ähnlichkeiten zu LISP aufweist, ist in meinen Augen gewöhnungsbedürftig und hat die bereits angesprochenen Nachteile einer proprietären Sprache. Zur Generierung der CASE-Werkzeuge bietet DOME zwei Varianten: eine Interpretation und ein Kompilieren des Metamodells. Da in letztem Fall Smalltalk als Zielsprache gewählt wurde, bietet sich hier theoretisch die Möglichkeit auf eine allgemeine Programmiersprache auszuweichen [EnK00].

Wie schon Tabelle 5-2 zu entnehmen ist, bietet MetaEdit+ keine solche Transformationssprache an. Denkbar ist allerdings eine Modifikation des Modells indem man die SOAP-Web-Services (Simple Object Access Protocol [Cer02]) für die Ankopplung eines externen Werkzeuges nutzt, das prinzipiell in jeder geeigneten Programmiersprache realisiert werden kann.

Auch GME weist keine eigene Transformationssprache auf und bietet daher einen externen Zugriff auf das Repository über das COM-Komponentenmodell (Component Object Model [Löw01]). Damit gilt das für MetaEdit+ Gesagte. Da GME XML-Dateien sowohl lesen als auch schreiben kann, existiert hier aber zusätzlich die Möglichkeit auf der Basis dieser Dateien die Modelltransformation durchzuführen. Zuletzt kann GME aus den Modelldaten C++-Code zu erzeugen, der eine Programmierschnittstelle für den Zugriff auf die Daten erlaubt [LMB01].

GME und DOME nutzen die reflexive Eigenschaft der Metamodellierung insofern, als dass das Meta-Metamodell der Werkzeuge selbst mit dem Werkzeug beschrieben wird und daraus dann das MetaCASE-Werkzeug generiert werden kann. Dies entspricht einem „Bootstrapping"-Schritt wie man ihn auch von Compilern kennt. In MetaEdit+ ist diese Vorgehensweise nicht möglich, da bei diesem Werkzeug die Metamodelle textuell (mit Hilfe von Formularen) erstellt werden.

Da alle vorgestellten MetaCASE-Werkezeuge vornehmlich für die Erstellung grafischer CASE-Werkzeuge gedacht sind, ist eine Bearbeitung textueller oder tabellenartiger Entwicklungsinformationen (wie z.B. Listen von Entwickleranforderungen) aufwändiger, da diese dann in einer grafischen Form editiert werden müssten.

Ein letzter Punkt, für den alle diese Werkzeuge keine Lösung anbieten, ist dass kein dokumentzentriertes und damit entkoppeltes Arbeiten möglich ist (siehe Abschnitt 5.1 zu Beginn dieses Kapitels). So bietet z.B. DOME nur einen eingeschränkten Mechanismus zur Aufteilung von Modellen in einzelne Teile [EnK00]. GME bietet zumindest eine durchgängige Versionierung der Modelle an.

5.3.4 Angrenzende Bereiche

Neben den oben vorgestellten Alternativen im Bereich der Modelle und Werkzeuge soll zum Ende eine kurze Übersicht über angrenzende Bereiche gegeben werden, die mehr oder weniger mit der Thematik der Automatisierung zusammenhängen.

Generative Programmierung

Die *generative Programmierung* ist in das Gebiet des Produktlinienentwurfs einzuordnen, bei welchem durch die Definition sog. Variationspunkte in einer gegebenen Menge von Produkt-Features ein Effizienzgewinn bei der Entwicklung ähnlicher Produkte erzielt wird [CzE00, S.36]. Unter Features (oder Produktmerkmalen) versteht man hierbei auffällige, kennzeichnende und unabhängige Teile, Aspekte, Charakteristiken oder Qualitäten von Software-Systemen [SRP03][PSC01, S.1ff.].

Der Schlüssel zur Automatisierung liegt dabei in der Definition eines (*generativen*) *Domänenmodells* während des „Domain-Engineerings", das aus einem Pro-

blem- und einem Lösungsraum besteht zwischen denen eine Abbildung auf der Basis von Konfigurationswissen hergestellt wird [Cza03].

Im „Application Engineering“ wird eine konkrete Konfiguration durch die Auswahl einer oder mehrerer Ausprägungen für die einzelnen Variationspunkte definiert [BKP04, S. 7] und damit die gewünschte Produktvariante (automatisch) abgeleitet. Für eine detailliertere Beschreibung dieses Ansatzes sei hier auf die Bücher von Czarnecki und Eisenecker [CzE00] und Böckle et al. [BKP04] verwiesen.

Domain Specific Languages

Eng mit der generativen Programmierung hängt die Technik der *domänenspezifischer Sprachen* (engl. „Domain Specific Languages“, DSL) zusammen. Neben der Definition einer für die Domäne angebrachten Modellierungsnotation wird bei einer DSL großes Gewicht auf die Generierung von effizientem, anwendungsspezifischem Produkt-Code gelegt. Deswegen existiert neben einem domänenspezifischen Metamodell zur Sprachbeschreibung auch ein angepasster Code-Generator, der sehr starken Bezug zu einem existierenden Satz von Komponenten (entstanden in einem Domain-Engineering-Schritt) aufweist [CzE00, S. 137ff.].

MetaCASE-Werkzeuge (siehe vorhergehender Abschnitt) wie GME und MetaEdit+ sind typische Werkzeuge für den Umgang mit DSLs, weshalb diese auch entsprechende Dokument- und somit Code-Generatoren mitbringen.

Metaprogrammierung

Ein Metaprogramm dient zur Manipulation (und vor allem auch Konstruktion) von Programmen. Es ist also ein Programm, das Programme erzeugt. Damit fallen in diese Klasse von Programmen typischerweise die Hochsprachen-Compiler (diese Erzeugen ein Programm in der Maschinensprache des Rechners) aber auch Theorembeweiser und Programmanalysierer und -transformatoren [She01].

Die Arbeiten zur Modelltransformation fallen prinzipiell in diese Klasse nur dass die erzeugten Artefakte keine Programme sondern Modelle sind. Überträgt man das Konzept vollständig auf Modelle, dann müsste auch die Transformation durch ein Modell beschrieben werden. Auf der Basis der in dieser Arbeit vorgestellten Aktionssprache AL++ (Abschnitt 4.2.2 auf Seite 65) wäre eine solche Abstraktion zum Beispiel zu erreichen und man hätte damit die Technik der „Metamodellierung“ formuliert, also die Spezifikation von Modellen zur Erzeugung von Modellen (der Begriff „Metamodellierung“ ist allerdings bereits für die Bezeichnung der Aktivität der Erstellung von Metamodellen belegt, Abschnitt 3.2 auf Seite 38).

Mehrebenen/-stufen Programmierung

Wenn es möglich ist, neben der Erzeugung von Programmcode durch ein Metaprogramm, diesen Code auch zur Laufzeit zu übersetzen und auszuführen, dann kann man die Metaprogrammierung mehrstufig durchführen. Handelt es sich bei dem vom Metaprogramm erzeugten Code wieder um ein Metaprogramm, so bezeichnet man diese Technik als *Mehrebenenen-Programmierung*. Einer der bekanntesten Vertreter einer solchen „Multi-Stage"-Programmiersprache ist MetaML [She01].

Generische Programmierung

Grundgedanke bei der *generischen Programmierung* ist, Software-Komponenten so zu konstruieren, dass diese nur minimale Annahmen über andere Komponenten machen und somit eine größtmögliche Flexibilität bei der Zusammensetzung der Komponenten besteht [DeS00].

Anwendung fand dieses Paradigma bei der Erstellung der C++ Standard Template Library (STL), welche auf dem C++ Template-Mechanismus basiert [Aus99]. Ein ähnlicher Mechanismus existiert für die Programmiersprache Java seit J2SE 1.5 unter dem Namen „*Generics*" [Aus04].

Generics bieten die Möglichkeit, Typen durch Angabe von Typen zu parametrisieren. Diese werden daher auch parametrisierte Typen genannt. Typisches Beispiel für einen parametrisierten Typ ist eine Collection (z.B. eine sortierte Menge), die nur Einträge eines bestimmten Typs T beinhaltet. Eine Definition einer solchen Liste erfolgt z.B. mit `SortedSet<T>`. Der Vorteil ist nun, dass bei einem Zugriff auf diese Menge stets Instanzen vom Typ T zurückgeliefert werden und man nicht einen expliziten Type-Cast durchführen muss. Damit handelt es sich bei einer solchen sortierten Menge um ein generisches *Konzept* [DeS00], das auf alle Typen angewendet werden kann, für die ein Vergleichsoperator (zum Sortieren) definiert ist.

Zusammenfassung

In diesem Kapitel wurde gezeigt, wie die Grundlagen aus den ersten Kapiteln eingesetzt werden können, um eine systematische Automatisierung von Software-Entwicklungsaktivitäten zu erreichen. Wichtig dabei war eine Trennung in konkrete und abstrakte Syntaxelemente, die Aufteilung von Verantwortlichkeiten auf die Werkzeuge und innerhalb dieser Werkzeuge auf die aus den Metamodellen abgeleiteten Klassen.

Außer der strukturellen Abbildung von Metamodellen auf Werkzeuge wurde auch gezeigt, wie unsere Aktionssprache AL++ auf Java abgebildet werden kann und wie man mit Hilfe eines einfachen Zwischenformats die Hintereinanderausführung der Automatisierungswerkzeuge vereinfachen kann.

Ein Vergleich mit verwandten Arbeiten, zu denen CASE-Werkzeuge, Meta-Metamodelle und Transformationsansätze gehören, schloss dieses Kapitel ab.

6 Anwendung auf eine existierende Entwicklungsmethode

Proband (lat. probandus: einer, der untersucht werden muss)
Bezeichnung für eine gesunde oder in Bezug auf das zu testende
Präparat oder Verfahren nicht einschlägig kranke Versuchsperson.

— [Pschyrembel – Klinisches Wörterbuch, 1998]

Als Übergang zum zweiten Teil dieser Arbeit, in welchem konkrete Automatisierungsaktivitäten vorgestellt werden, wollen wir in diesem Kapitel die bisher erarbeiteten Grundlagen und Techniken auf eine bestehende Methode anwenden. Dazu wird im Folgenden die in unserer Arbeitsgruppe entwickelte PROBAnD-Methode vorgestellt und deren Produktmodell als Instanz unseres Produktmetamodells (siehe vorheriges Kapitel) interpretiert.

Nach der Beschreibung der allgemeinen Vorgehensweise der PROBAnD-Methode und deren Illustration an einem überschaubaren Beispiel folgt die Definition der abstrakten und konkreten Syntax der Entwicklungsartefakte. Zum Abschluss wird an einem kleinen Beispiel gezeigt, wie eine einfache Modellierungsaktivität der PROBAnD-Methode automatisiert werden kann.

6.1 Die PROBAnD-Methode

Bei der *PROBAnD*-Methode (*Prototyping, Reuse- and Object-based Building Automation Development* [Que02][MeQ02][MeQ01]) handelt es sich um eine domänenspezifische Software-Entwicklungsmethode, welche zunächst für die Analysephase in der Entwicklung reaktiver Systeme mit statischer Struktur konzipiert wurde. Das ursprüngliche Anwendungsgebiet dieser Methode war die Gebäudeautomation (vgl. [Kra97]). In mehreren Fallstudien wurde jedoch auch deren Anwendbarkeit auf Au-

tomotive-Controller [MeZ03] und deren Erweiterbarkeit auf Systeme mit dynamischer Struktur (z.B. Steuerung einer Eisenbahnkreuzung [QuM02]) gezeigt.

Interessant ist die PROBAnD-Methode für die Untersuchungen in dieser Arbeit nicht nur, weil sie bereits mehrfach erfolgreich in unserer Arbeitsgruppe eingesetzt wurde, sondern vor allem auch deshalb, weil notwendige Änderungen oder Ergänzungen der Methode einfach durchführbar sind. Insbesondere von der Erweiterbarkeit der Methode wurde Gebrauch gemacht (siehe dazu Abschnitt 6.2.3 auf Seite 151). Die PROBAnD-Methode ist im Kontext dieser Arbeit also hauptsächlich als Testvehikel zu verstehen. Deshalb werden wir an dieser Stelle die Vorzüge und Ansätze von PROBAnD nicht detailliert diskutieren, sondern verweisen hierzu auf die Arbeit von Queins [Que02].

Wir sind uns allerdings sicher, dass die mit PROBAnD erzielten Ergebnisse auch auf andere Entwicklungsmethoden übertragbar sind. Dies lässt sich schon daraus ableiten, dass verwandte Methoden an vielen Stellen Gemeinsamkeiten mit PROBAnD aufweisen. So erfolgt z.B. in *ROPES* (*Rapid Object-Oriented Process for Embedded Systems* [Dou99]) eine ereignis- und zustandsorientierte Modellierung (mit der UML) oder in *TIMe* (*The Integrated Method* [BGH99]) die Spezifikation komplexer reaktiver Systeme mit Hilfe der SDL-Notation [OFM94].

6.1.1 Reaktive Systeme

Die PROBAnD-Methode wird für die Spezifikation von reaktiven Systemen und insbesondere von *Mess-*, *Steuer-* und *Regelsystemen* (*MSR-Systemen*) eingesetzt.

Typische MSR-Systeme sind z.B. Prozess-, Gebäude- [BCG01] und Heimautomationssysteme [Har03] und Automotive-Controller, wie z.B. ABS- oder ESP-Systeme [Saa03]. In diesem Abschnitt wollen wir daher zunächst die wichtigsten Begriffe aus der Domäne der reaktiven Systeme klären.

Zeitliches Verhalten

Eine wichtige Eigenschaft der reaktiven Systeme, die sie deutlich von Transformationssystemen unterscheidet ist, dass reaktive Systeme normalerweise nie terminieren, sondern kontinuierlich auf *Stimuli* aus ihrer Umgebung reagieren [MaP96, S.2]. Dies hat zur Folge, dass die Korrektheit eines eingebetteten Systems nicht mit dem traditionellen Korrektheitsbegriff von Algorithmen vereinbart werden kann, da dieser die Terminierung und die Rückgabe eines Resultats voraussetzt (siehe dazu z.B. [Bak80, S.1]). Weitere Probleme bei der Verifikation reaktiver Software werden von Cousot et al. in [CoC01] diskutiert.

Obwohl man ein reaktives System wegen dessen Interaktion mit seiner Umgebung als *interaktives System* einordnen könnte, weisen reaktive und interaktive Sys-

teme einen fundamentalen Unterschied bzgl. der Synchronisationsmechanismen auf. Bei interaktiven Systemen besitzt sowohl die Umgebung als auch das System Synchronisationsmöglichkeiten. So kann z. B. bei einem interaktiven Grafikeditor der Benutzer auf die Bearbeitung einer Grafikoperation warten. Dies ist bei einem reaktiven System nicht möglich. Systeme wie *ESP* (*Electronic Stability Program* [BDC00, S. 701]) oder Notabschaltungen technischer Prozesse müssen in einer Gefahrensituation sofort reagieren. Dies impliziert für ein reaktives System, dass

1. das System stets auf Stimuli der Umgebung reagieren können muss und
2. die Reaktion des Systems so schnell erfolgen muss, dass die Umgebung für diese Reaktion noch „empfänglich“ ist.

Diese beiden Punkte stellen Anforderungen an das Echtzeitverhalten des Systems dar. Daher werden reaktive Systeme oft als *Echtzeitsysteme* charakterisiert, wodurch Gewichtung auf diesen Zeitaspekt gelegt wird [MaP96, S. 2ff.].

Einbettung

Neben diesen zeitlichen Eigenschaften zeichnen sich reaktive Systeme dadurch aus, dass sie in einer physikalischen *Umgebung* eingebettet sind (die Interaktion mit dieser Umgebung wurde oben bereits angesprochen). Die Rolle der Umgebung bei reaktiven Systemen ist in Abb. 6-1 grafisch dargestellt.

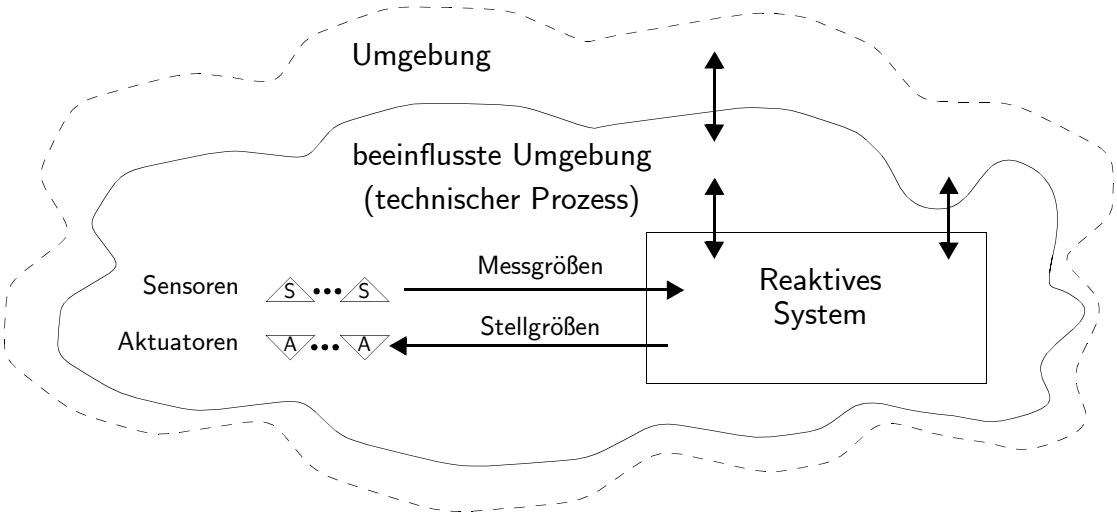


Abbildung 6-1. Die Rolle der Umgebung bei reaktiven Systemen

Diese Umgebung wollen wir etwas genauer betrachten. Aus der Sicht des reaktiven Systems ist zunächst der beeinflusste Bereich (der technische Prozess) relevant. Für eine Lichtregelung bedeutet dies z. B., dass die Helligkeit der Umgebung des reaktiven Systems eine relevante Größe darstellt nicht jedoch die Temperatur im gesteuerten Raum.

Zur Erfassung der relevanten Messgrößen dienen *Sensoren*, welche die Schnittstelle zwischen Umgebung und System darstellen. Wo die genaue Grenze zwischen Umgebung und reaktivem System zu ziehen ist, ist allerdings abhängig von der Perspektive und dem Modellierungszweck. Queins stellt in [Que02, S. 181ff.] eine Aufteilung von Sensoren in fünf Schichten dar, wovon unserer Ansicht nach die Grenze zwischen jeder dieser Schichten angesiedelt sein könnte.

Um die Umgebung im gewünschten Maße zu beeinflussen werden *Aktuatoren* (*Stellglieder*) eingesetzt. Auch diese sind Teil der Schnittstelle zwischen Umgebung und reaktivem System. Für die Definition der Grenze gelten daher analoge Betrachtungen.

Wie man in Abb. 6-1 erkennt, ist die beeinflusste Umgebung selbst wieder Teil einer umfassenderen Umgebung. Der beeinflusste Teil kann daher selbst wieder von dessen Umgebung beeinflusst werden (angedeutet durch die vertikalen Pfeile). Als Beispiel sei hier das Wetter genannt, welches nicht beeinflussbar ist, aber die Helligkeit in einem Raum (abhängig von Bewölkung, Tageszeit, etc.) spürbar beeinflusst. Desweiteren können Einflüsse der Umgebung auf das reaktive System außerhalb der Sensor-/Aktuatorschnittstelle einwirken. So kann die Raumtemperatur die Betriebstemperatur des Hardware-Controllers, auf welchem die Steuerungs-Software läuft, in einem solchen Maße beeinflussen, dass der zulässige Betriebsbereich überschritten wird.

Neben der Einbettung eines reaktiven Systeme in dessen Umgebung ist auch – wie im vorhergehenden Absatz angedeutet – dessen Einbettung in Hardwarekomponenten möglich. So handelt es sich z.B. bei Mobilfunktelefonen, CD-Spielern und ähnlichen Geräten um solche *eingebetteten Systeme* (engl. „embedded systems“).

Konsequenzen für eine Entwicklungsmethode

Die obigen Eigenschaften reaktiver Systeme beeinflussen die Wahl der Vorgehensweise und der Notationen (Abstraktionen) bei der Definition einer Entwicklungsmethode. So bedeutet die reaktive Eigenschaft, dass das Systemverhalten sinnvollerweise auf der Basis von Ereignissen spezifiziert werden sollte. Desweiteren ist häufig eine zustandsorientierte Spezifikation angebracht, da das Verhalten vom jeweils aktuellen (und sichtbaren) Systemzustand abhängt.

Das Vorhandensein der Umgebung und speziell die explizite Schnittstelle über Sensoren und Aktuatoren erfordern deren Berücksichtigung schon bei der Spezifikation des reaktiven Systems. Bei der Verifikation und Validierung spielt die Umgebung eine sehr wichtige Rolle, da sinnvolle Tests nur im Zusammenspiel des Systems mit seiner Umgebung durchgeführt werden können.

Da man von der Existenz der Umgebung zu Beginn der Entwicklung des reaktiven Systems nicht ausgehen kann (normalerweise wird die Umgebung, z.B. ein Ge-

bäude, parallel zur Entwicklung des Systems realisiert), ist es notwendig neben der reinen Systemspezifikation auch eine Modellierung der Umgebung umzusetzen, aus welcher Umgebungssimulatoren abgeleitet werden können. Riegel stellt z. B. für diese Zwecke in [Rie04] ein Methode zur musterbasierten Entwicklung von Gebäudesimulatoren vor.

Der Fokus der im Folgenden vorgestellten PROBAnD-Methode liegt zunächst auf der Spezifikation reaktiver Systeme. Allerdings wurde eine Erweiterung dieser Methode auch erfolgreich für die Spezifikation von Umgebungssimulatoren – im Speziellen für die Realisierung von Gebäudesimulatoren – angewendet (siehe [Zim02] und [MMZ02]).

6.1.2 Produkte und Aktivitäten

Nach der Einordnung der PROBAnD-Methode in die Domäne der reaktiven Systeme soll in diesem Abschnitt zunächst ein Überblick über die Produkte und Aktivitäten der Methode gegeben werden (siehe dazu Abbildung 6-2), was das Verständnis der in Abschnitt 6.2 auf Seite 148 präsentierten Artefakte des Produktmodells erleichtern wird.

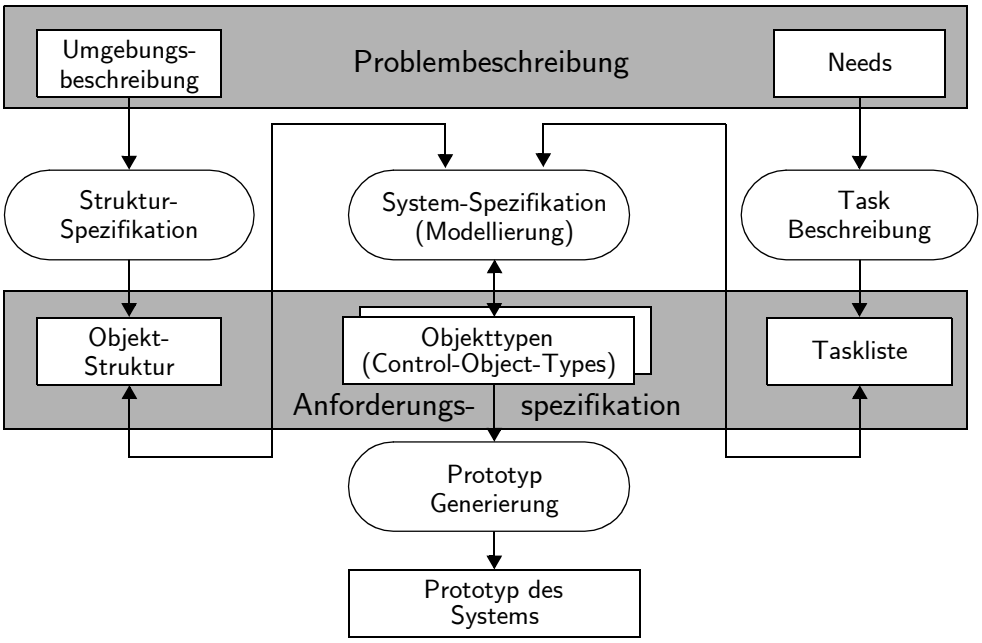


Abbildung 6-2. Übersicht über Dokumenttypen und Aktivitäten der PROBAnD-Methode

Die Spezifikation (oder auch Modellierung) eines reaktiven Systems erfolgt ausgehend von einer *Problembeschreibung*, welche aus einer *Umgebungsbeschreibung* und einer Auflistung der (funktionalen) *Benutzeranforderungen* (*Needs*) besteht.

Die Umgebungsbeschreibung beinhaltet mindestens eine Beschreibung der Struktur der Umgebung. Im Falle eines Gebäudeautomationssystems könnte diese Struk-

turbeschreibung z. B. durch die Darstellung des Gebäudegrundrisses erfolgen. Neben einer solchen Darstellung von Räumen und Flursegmenten beinhaltet die Umgebungsbeschreibung i. d. R. zusätzlich eine Auflistung der vorhandenen (oder zu installierenden) Sensoren und Aktuatoren.

Aus dieser Umgebungsbeschreibung wir eine initiale *Objektstruktur* (oder *Modellarchitektur*) abgeleitet, welche in späteren Entwicklungsschritten verfeinert werden kann. Wie Rausch et al. in [RaB02] herausstellen, ist eine solche „Software-Architektur" ein „kritischer Erfolgsfaktor". Nur durch eine gute Architektur kann ein Software-System von angemessener Qualität erreicht werden, da erst hiermit eine leichte Änderbarkeit der Spezifikation, deren sinnvolle Wiederverwendung [Sie04, S. 1ff.], deren Einsatz als Kommunikationsgrundlage und das Beherrschen von Komplexität [RaB02] möglich werden. Auch in dieser Arbeit werden wir an vielen Stellen von der Modellarchitektur der PROBAnD-Methode Gebrauch machen.

In der Objektstruktur werden dazu zunächst *Control-Object-Types* als Klassifikation der Objekte eines reaktiven Systems definiert, um die Vielzahl an Objekten, die man bei der Entwicklung eines solchen Systems berücksichtigen muss, beherrschen zu können. Dabei besitzt jedes Control-Objekt (also jede Instanz eines Control-Object-Types) einen unabhängigen Kontrollfluss und ist daher echt konkurrent zu anderen Control-Objekten, was die parallele Natur der Realweltobjekte (z. B. der physikalischen Sensoren und Aktuatoren) widerspiegelt.

Zwischen diesen Control-Object-Types werden Aggregationsbeziehungen so festgelegt, dass zur Laufzeit des Systems ein Baum von Control-Objekten instanziiert wird. Diese Baumstruktur entspricht der Struktur der Umgebung, was sich in den mit der PROBAnD-Methode bisher bearbeiteten Domänen als erfolgreicher Ansatz zur Definition der Modellarchitektur bewährt hat. Der Grund liegt darin, dass sich viele der Control-Objekte mit den Objekten in der Umgebung identifizieren lassen und diese auch typischerweise baumförmig aggregiert werden können (z. B. besteht ein Gebäude aus Stockwerken und diese wiederum aus Räumen).

Neben der Festlegung der statischen Struktur dienen die Aggregationsbeziehungen auch der Definition von zulässigen Kommunikationskanälen zwischen Control-Objekten. In den PROBAnD-Modellen erfolgt eine solche Kommunikation mittels asynchroner Nachrichten (Signale), die ausgehend von einem Control-Objekt entweder zu dem aggregierenden Control-Objekt oder zu einem oder mehreren aggregierten Control-Objekten gesendet werden können.

Ausgehend von dem anderen Teil der Problembeschreibung, den Needs, welche die Benutzeranforderungen an das reaktive System beinhalten, werden *Entwickleranforderungen* (*Tasks*) abgeleitet. Sowohl Needs als auch Tasks werden natürlichsprachlich beschrieben, wobei eine formale Verfolgbarkeitsrelation von Needs zu Tasks und zu eventuellen Sub-Tasks besteht.

Wichtige Vorgabe bei der Formulierung der Entwickleranforderungen ist, dass jeder Task genau *einem* Control-Object-Type zugeordnet wird. Durch diese eindeutige Zuweisung können die Tasks auch als Verantwortlichkeiten der einzelnen Objekttypen interpretiert werden. Diese Einschränkung, welche die Automatisierung von Entwicklungsaktivitäten an vielen Stellen vereinfacht (siehe zweiter Teil der Arbeit), hat sich in der Praxis als nicht problematisch erwiesen und auch komplexe Systeme (siehe z.B. Abschnitt 8.2.2 auf Seite 224) konnten mit Hilfe der PROBAnD-Methode spezifiziert werden.

Das Finden von Tasks und die Zuweisung zu eventuell neu zu definierenden Control-Object-Types ist ein iterativer Prozess, d.h. dass sich durch die Formulierung neuer Tasks auch neue Objekttypen ergeben können und umgekehrt.

Nachdem die wichtigsten Control-Object-Types identifiziert wurden und diesen Objekttypen Tasks zugeordnet wurden, beginnt man mit der Beschreibung sog. *Strategien*, welche zunächst eine natürlichsprachliche Beschreibung der Realisierung der Tasks beinhalten und bei einer verfeinerten Beschreibung um eine operationale Verhaltensspezifikation in der Form von (partiellen) Zustandsautomaten ergänzt werden können (die exakte Beschreibung dieser Darstellung wird in Abschnitt 6.2.3 auf Seite 151 eingeführt). Vereinfacht wird eine solche Verhaltensbeschreibung durch die Tatsache, dass die Control-Object-Types gerade so gewählt wurden, dass sie eine Entsprechung in der Realität besitzen. Daher lassen sich auch die Zustände solcher Objekttypen sehr einfach in Analogie zu den Realweltobjekttypen ableiten. Eine Leuchte (Aktuator) hat so z.B. die Zustände „an“ und „aus“, ein Controller zur Helligkeitsregelung die möglichen Zustände „zu hell“, „ok“ und „zu dunkel“.

Bis zu dieser Aktivität werden die Entwicklungsinformation in der Form von strukturierten Tabellen mit Hilfe von HTML beschrieben (in Abschnitt 6.1.3 und in [Que02, S.71ff.] ist eine beispielhafte Beschreibung dieser Dokumente zu finden). Ab diesem Punkt werden nun zusätzlich Dokumente in der SDL-Notation [OFM94] beschrieben, womit eine Generierung von Prototypen mit Standardwerkzeugen (in unserem Fall mit der Telelogic Tau SDL Suite [Dol03][LEH00]) möglich wird. Die automatische Abbildung von HTML nach SDL wird in den Kapiteln 7 und 9 vorgestellt.

Mit der Existenz von Prototypen ist nun auch eine Verifikation und Validierung der Spezifikation möglich. Zusammen mit der Inspektion der Entwicklungsdokumente (siehe [Que02, Kapitel 8]) dienen diese Maßnahmen der Qualitätssicherung. Im zweiten Teil dieser Arbeit werden wir im Kontext der Automatisierung von Software-Entwicklungsaktivitäten darauf genauer eingehen.

Es soll an dieser Stelle noch angemerkt werden, dass die Objektstruktur und dementsprechend die Control-Object-Types während der Anforderungsanalyse, bzw. -spezifikation zunächst für eine Strukturierung der Spezifikation, zum leichten Verständnis und der einfachen Orientierung im Modell, eingeführt wird. Während

der Entwurfsphase (Design) ist eine Änderung der Struktur auf Grund von Design-Entscheidungen möglich. Desweiteren sollte die operationale Spezifikation der Strategien zunächst nur als *mögliche* Realisierung betrachtet werden, welche für ein frühes Prototyping notwendig ist. Während der Entwurfsphase können hier alternative Realisierungen betrachtet oder sogar notwendig werden.

6.1.3 Ein Beispiel

Um die Anwendung der PROBAnD-Methode für die Spezifikation reaktiver Systeme zu veranschaulichen, wollen wir hier ein Beispiel aus der Domäne der Gebäudeautomation vorstellen, welches die Spezifikation eines Automationssystems für einen Raum beinhaltet. Neben der Vermittlung eines Eindrucks der verschiedenen Dokumenttypen wird uns dieses Beispiel auch im weiteren Verlauf der Arbeit zur Illustration der einen oder anderen Automatisierungsaktivität dienen, weshalb wir dieses System *RoomAutomation* nennen wollen.

Problembeschreibung

Zunächst betrachten wir die als Ausgangspunkt der Entwicklung gegebene Problembeschreibung. Abb. 6-3 zeigt die Gebäudestruktur und die Platzierung von Sensoren und Aktuatoren.

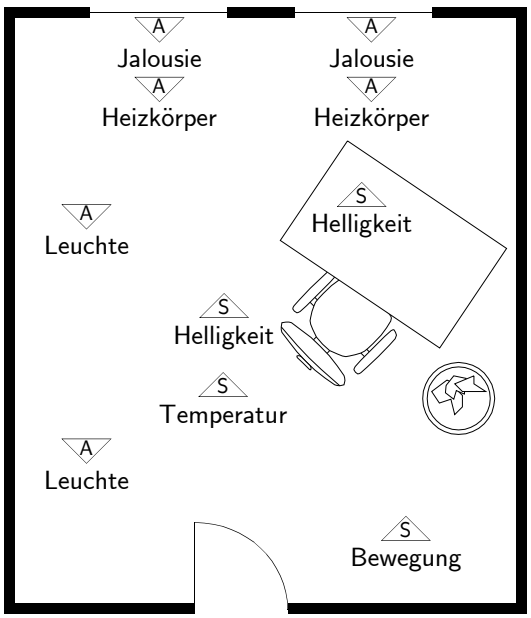


Abbildung 6-3. Struktur des zu automatisierenden Raums als Teil der Problembeschreibung

Desweiteren wollen wir annehmen, dass die in Tabelle 6-1 gegebenen vier Benutzeranforderungen formuliert wurden. Auch wenn die echten Entwicklungsdokumen-

te durch HTML-Dokumente beschrieben werden, wählen wir in dieser Arbeit der Lesbarkeit halber normale Tabellen. Die dargestellte Information ist identisch.

Die erste Anforderung N1 drückt den Benutzerwunsch nach einer automatischen Helligkeitssteuerung aus. Der zweite Need (N2) spiegelt ein Energieoptimierungskriterium wider (also quasi eine nicht-funktionale, oder „Quality-of-Service"-Anforderung). Um ungestört arbeiten zu können wurde schließlich Anforderung N3 spezifiziert. Zuletzt wird in N4 eine Anforderung an eine Temperaturregelung formuliert.

Tabelle 6-1. Benutzeranforderungen für *RoomAutomation*

Need	*Description*
N1	Für die geforderte Beleuchtung soll gesorgt werden.
N2	Durch die Verwendung von Tageslicht soll Energie eingespart werden.
N3	Eine Blendung am Arbeitsplatz ist zu vermeiden.
N4	Für die geforderte Temperatur in einem Raum soll gesorgt werden.

Objektstruktur

Die aus der Umgebungsbeschreibung abgeleitete und durch die Aufstellung der Tasks (siehe unten) verfeinerte Objektstruktur ist in Abb. 6-4 gezeigt.

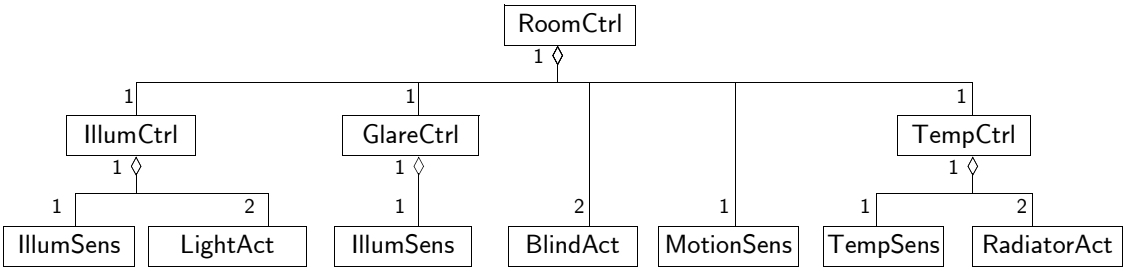


Abbildung 6-4. Objektstruktur eines einfachen Gebäudeautomationssystems

Der oberste Control-Object-Type RoomCtrl wurde aus dem Umgebungsobjekttyp „Raum" abgeleitet. Desweiteren erkennt man Sensoren und Aktuatoren, die typischerweise als Blätter im Objektgraphen auftauchen. Dazu gehören IllumSens (Helligkeitssensor), LightAct (zum Schalten der Leuchten), BlindAct (zum Fahren der Jalousien), MotionSens (Bewegungsmelder), TempSens (Temperaturmessung) und RadiatorAct (zum Öffnen oder Schließen der Heizkörperventile). Die zwei Helligkeitssensoren werden eingeführt, weil der eine der Messung der Raumhelligkeit und der andere der Feststellung der Helligkeit am Arbeitsplatz dient (siehe auch deren Platzierung in Abb. 6-3).

Als Verfeinerung dieser initial von der Raumbeschreibung abgeleiteten Struktur wurden die Knoten IllumCtrl, GlareCtrl und TempCtrl eingeführt, welche für die Regelung oder Steuerung der jeweiligen physikalischen Effekte zuständig sind.

Tabelle 6-2 zeigt, wie die Objektstruktur in einem entsprechenden HTML-Dokument der PROBAnD-Methode beschrieben wird.

Tabelle 6-2. Objektstruktur in Tabellenform

ControlObjectType	*InstanceName*	*Number*	*InstantiatedControlObjectType*
RoomCtrl	ic	1	IllumCtrl
	gc	1	GlareCtrl
	tc	1	TempCtrl
	ba	2	BlindAct
	ms	1	MotionSens
IllumCtrl	is	1	IllumSens
	la	2	LightAct
GlareCtrl	is	1	IllumSens
TempCtrl	ts	1	TempSens
	ra	2	RadiatorAct

Zusätzlich zu der Anzahl der jeweils aggregierten Instanzen wird darin noch der Basisname der Instanz (InstanceName) angegeben, an welchen jeweils die laufende Nummer der Instanz angehängt wird. Damit besteht die Möglichkeit alle Instanzen in der Spezifikation eindeutig zu benennen. Beginnend bei der Wurzel des Instanzenbaumes (stets obj1 genannt) werden die Namen der einzelnen Instanzen jeweils konkateniert. Für das *RoomAutomation*-Beispiel hätte so die Instanz des TempSens den eindeutigen Namen ts1.tc1.obj1.

Tasks (Entwickleranforderungen)

Nach der Beschreibung der Objektstruktur wollen wir uns nun eine mögliche Realisierung der Benutzeranforderungen durch Tasks anschauen. Dazu sind diese in Tabelle 6-3 zunächst aufgelistet.

Tabelle 6-3. Entwickleranforderungen (Tasks) von *RoomAutomation*

Task	*Description*	*realizes*	*implementedBy*
T1	Wenn der Raum besetzt ist, die Innenhelligkeit mit den zur Verfügung stehenden Lichtquellen (Jalousie und Leuchte) unter Berücksichtigung des Energieverbrauchs regeln.	N1, N2	IllumCtrl
T2	Auf Anforderung das Licht ein- oder ausschalten.	T1	LightAct
T3	Auf Anforderung die Jalousien öffnen oder schließen.	T1, T6	BlindAct
T4	Bewegung feststellen und melden.	T1, T6	MotionSens
T5	Aktuelle Helligkeit messen und auf Anforderung melden.	T1, T6	IllumSens
T6	Blendung am Arbeitsplatz durch Einsatz der Jalousie vermeiden.	N3	GlareCtrl

Tabelle 6-3. Entwickleranforderungen (Tasks) von *RoomAutomation*

Task	*Description*	*realizes*	*implementedBy*
T7	Raumtemperatur mit Heizkörper regeln.	N4	TempCtrl
T8	Auf Anforderung Heizkörperventil öffnen oder schließen	T7	RadiatorAct
T9	Aktuelle Temperatur messen und auf Anforderung melden.	T7	TempSens

Die Tasks der Sensoren und Aktuatoren sind sehr einfach, da diesen Objekttypen nur einfache Aufgaben obliegen. Komplexer sind die Tasks für die eigentlichen Controller (Light-, Glare- und TempCtrl). Diese können ihre Anforderungen nur unter Zuhilfenahme anderer Objekttypen (bzw. deren Tasks) realisieren. Als Beispiel sei hier der Task T7 der Temperaturregelung TempCtrl genannt, welcher von T8 und T9 realisiert wird.

Weiter erkennt man an der Task-Beschreibung die gewählte Realisierung der Benutzeranforderungen (Needs). Für die Lichtsteuerung wird z.B. angegeben, welche Lichtquellen benutzt werden und auch, dass Licht nur dann eingeschaltet werden soll, wenn sich jemand im Raum befindet (Need N2).

Control-Object-Types (HTML)

Nach der Festlegung der initialen Tasks folgt die Spezifikation der Control-Object-Types. Diese beinhaltet verschiedene Aspekte: die Einordnung in die Aggregationsstruktur, die Definition von Strategien und schließlich die Spezifikation von Attributen, Signaltypen, und Timern. In den folgenden Tabellen werden diese für den Objekttyp TempCtrl beispielhaft vorgestellt. Alle diese Teiltabellen sind jeweils Bestandteil eines einzelnen HTML-Dokuments zur Beschreibung eines Control-Object-Types.

Zunächst werden für jeden Objekttyp die von ihm aggregierten Instanzen spezifiziert (Tabelle 6-4).

Tabelle 6-4. Beispielhafte Instanziierungstabelle (TempCtrl)

InstanceName	*Number*	*InstantiatedControlObjectType*
ts	1	TempSens
ra	2	RadiatorAct

Wie man erkennt stellt diese Teiltabelle einen Ausschnitt aus der Objektstruktur aus Tabelle 6-2 dar.

Als Nächstes erfolgt die Spezifikation der Strategien, also eine Verfeinerung der Tasks (Tabelle 6-5). Neben einer natürlichsprachlichen Beschreibung einer Strategie kann diese auch mit Hilfe partieller Zustandsautomaten (siehe dazu Abschnitt 6.2.3 auf Seite 151) beschrieben werden. Desweiteren kann eine Strategie die Defi-

nition weiterer Tasks erfordern, die zur Realisierung des beschriebenen Tasks beitragen.

Tabelle 6-5. Beispielhafte Tabelle zur Verfeinerung von Tasks (TempCtrl)

Task	*Description*	*Strategy Description*	*Transition*	*realizes*
T7	Raumtemperatur mit Heizkörper regeln.	Die Temperatur im Raum periodisch mit den Heizkörperventilen regeln (wenn zu kalt, Heizkörper öffnen; sonst schließen).	idle: ^newTemp(thetaIst)/ e := thetaSoll - thetaIst; !computePos; ^setRad(pos) -> ra1, ra2; :idle	N4
T7.1	Raumtemperatur prüfen.	Periodisch Raumtemperatur anfordern.	idle: @T1/ ^getTemp -> ts1; +T1(dCon); :idle	T7
T7.2	Initialisierung.	Initial Timer aufziehen.	: / ^getTemp -> ts1; +T1(dCon); :idle	T7

Für den bereits in der Task-Liste definierten Task T7 erfolgt die Wiederholung der entsprechenden Information (Name des Tasks, dessen Beschreibung und die realisierten Anforderungen). Zusätzlich erkennt man die natürlichsprachliche Formulierung der Strategien. Zur Realisierung von Task T7 werden zunächst zusätzliche Tasks (T7.1 und T7.2) definiert, welche zur periodischen Anforderung der Raumtemperatur (T7.1) und zur Initialisierung eines Timers zur periodischen Abfrage (T7.2) dienen.

Die Transition für T7 beinhaltet den eigentlichen Regelkreis. Ausgehend vom Zustand idle wird mit jedem Empfang einer neuen Temperatur (Signal newTemp) die Differenz („Error“) e der aktuellen Raumtemperatur thetaIst zum Sollwert thetaSoll berechnet (ϑ ist die Celsius-Temperatur, vgl. [LJK94, S.591]). Dann wird mit Hilfe der Prozedur computePos ermittelt, ob die Heizkörper geöffnet oder geschlossen werden sollen (der Prozedurrumpf von computePos ist nicht Teil des HTML-Dokuments). Abschließend wird diese Stellgröße per Signal setRad an die beiden Instanzen der Heizkörper (ra1 und ra2) gesendet.

Die Transitionen für T7.1 und T7.2 sind ähnlich, außer dass man mit @T1 auf das Timeout eines Timers wartet und mit +T1 diesen auf eine gewisse Zeitspanne dCon aufzieht (relativ).

Zur Realisierung der Tasks in Strategien bzw. Transitionen werden also Variablen (Attribute des Control-Object-Types), Signale (gekennzeichnet durch „^“), und

Timer (beschrieben durch die Symbole „@“ und „+“) verwendet. Diese werden in den folgenden Teiltabellen genauer definiert.

Tabelle 6-6. Beispielhafte Definition der Attribute (TempCtrl)

Attribute	*Type*	*Value*	*Tasks*	*Usage*	*Description*
e	Real		T7	read/write	Differenz Soll-/Ist-Wert
pos	Real		T7	read/write	Stellgröße
thetaIst	Real		T7	read/write	Ist-Temperatur
thetaSoll	Real	20.0	T7	read	Soll-Temperatur
dCon	Duration	10	T7.1, T7.2	read	Intervall zwischen Regelungen

Neben dem Namen eines Attributs werden in Tabelle 6-6 der Datentyp und eine mögliche Anfangsbelegung spezifiziert. Desweiteren kann angegeben werden, welche Tasks lesend („read“) bzw. schreibend („write“) auf dieses Attribut zugreifen, was wiederum der Verfolgbarkeit zwischen Anforderungen und Realisierung dient (siehe Abschnitt 6.1.4).

Tabelle 6-7. Beispielhafte Definition der Signaltypen (TempCtrl)

SignalType	*FormalParameter*	*Tasks*	*Usage*	*Description*
getTemp		T7.1, T7.2	produced	Temperaturanforderung
newTemp	Real	T7	consumed	Temperaturmeldung
setRad	Real	T7	produced	Ansteuerung Heizkörper

Zur Kommunikation mit anderen Objekttypen werden in Tabelle 6-7 Signaltypen definiert. Neben einem Namen wird ein solcher Signaltyp durch dessen formale Parameter beschrieben. Wie bei den Attributen wird auch hier wieder die Verfolgbarkeit zu den Tasks über die Usage-Spalte hergestellt. Erlaubt sind die Einträge „produced“ für produzierte und „consumed“ für konsumierte Signale.

Zuletzt werden in Tabelle 6-8 die verwendeten Timer definiert.

Tabelle 6-8. Beispielhafte Definition der Timer (TempCtrl)

Timer	*Tasks*	*Usage*	*Description*
T1	T7.2	modified	Timer zur periodischen Abfrage der Temperatur
	T7.1	modified/received	

Die Usage-Spalte beinhaltet analog zu den anderen Tabellen Informationen bzgl. der Verfolgbarkeit zu den Tasks. Erlaubt sind hier „modified“ für die Modifikation eines Timers (aufziehen oder zurücksetzen) und „received“ für das Empfangen eines Timeout-Ereignisses. Sind mehrere Angaben für einen Timer zu machen (wie in die-

sem Beispiel), so erfolgt dies, indem man in der folgenden Zeile die bereits definierten Daten (insbesondere den Namen) auslässt.

Control-Object-Types (SDL)

Als letzte Aktivität der Modellierung reaktiver System werden bei PROBAnD die HTML-Objekttypen nach SDL überführt (in Kapitel 9 werden wir erläutern wie dies automatisiert erfolgt). Bei einer manuellen Erstellung solcher Dokumente kommt einem hierbei eine Bibliothek von Vorlagen („Templates“) entgegen, die zwar eine strikte Form vorgeben, einem dadurch allerdings auch viele stupide Entwicklungsarbeiten abnehmen (weitere Details dazu sind in [MeQ03] zu finden).

Beginnen wir mit der Abbildung der strukturellen Modellinformationen auf SDL. Abb. 6-5 zeigt die Deklaration des Control-Object-Types TempCtrl als SDL Blocktyp (siehe [OFM94, S.119ff.]).

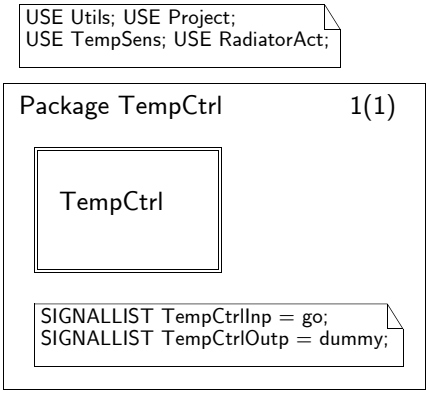


Abbildung 6-5. Beispielhafte Realisierung der Objektstruktur in SDL (Blocktypdeklaration)

Bei unserer Vorgehensweise definieren wir für jeden SDL-Blocktyp ein eigenes SDL-Package [OFM94, S.181f.]. Die Packages TempSens und RadiatorAct werden eingebunden, da die in diesen Packages definierten Blocktypen von TempCtrl aggregiert werden. (Die Packages Utils und Project beinhalten allgemeine bzw. projektspezifische Definitionen oder Hilfsfunktionen.)

Neben der Deklaration des Blocktyps (Symbol mit doppelter Umrandung) erkennt man auch die Definition zweier Signallisten TempCtrlInp und TempCtrlOutp. Dabei beinhaltet die Inp-Signalliste alle Signaltypen, die von den TempCtrl aggregierenden Objekttypen empfangen werden können. Die Outp-Signaltypen werden von TempCtrl an diese aggregierenden Objekttypen gesendet. In unserem Beispiel sind diese Listen quasi leer, da die Temperaturregelung noch unabhängig von dem restlichen *RoomAutomation*-System ist (insbesondere ist Need N2 noch nicht eingeflossen, weshalb die Temperaturregelung keine Rücksicht auf den Belegungszustand des Raumes nimmt).

Der Signaltyp dummy wird benötigt, um eine leere Liste definieren zu können (ohne die Angabe mindestens eines Signaltyps kann keine Liste definiert werden [OFM94, S.61]). Der Signaltyp go wird für die eindeutige Namensvergabe der Instanzen benötigt (siehe weiter oben) und ist daher in allen Blocktypen definiert.

In Abbildung 6-6 ist die Definition des Blocktyps TempCtrl zu sehen.

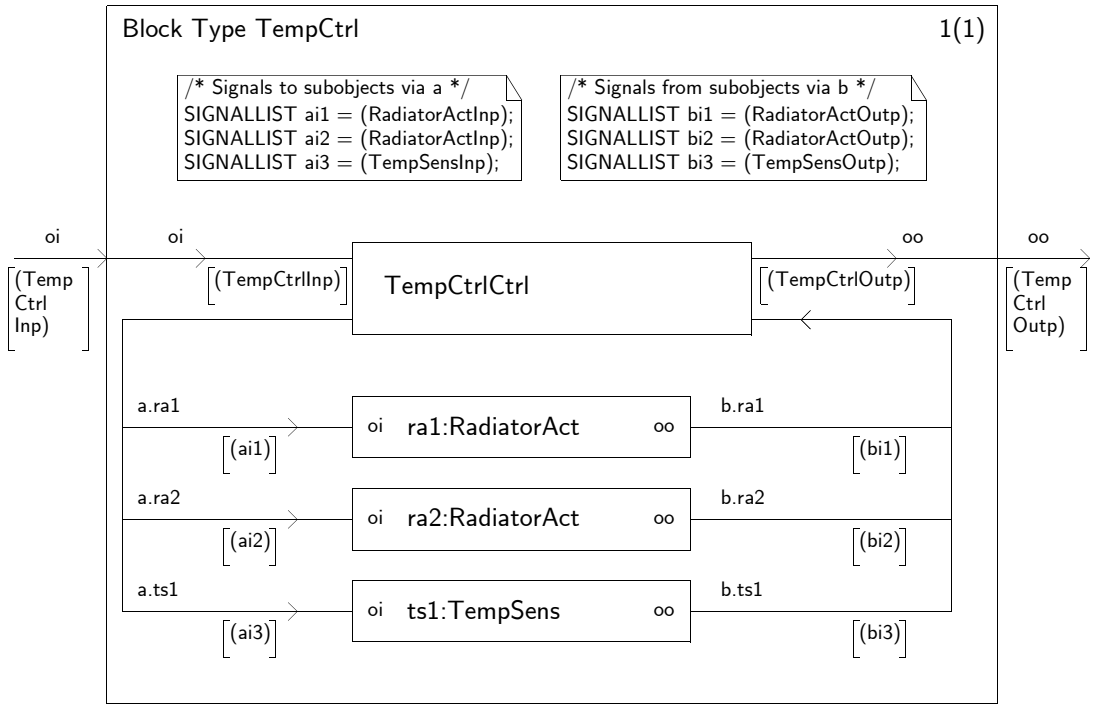


Abbildung 6-6. Beispielhafte Realisierung der Objektstruktur in SDL (Blocktypdefinition)

Der Blocktyp TempCtrl aggregiert eine Instanz des Blocktyps TempSens und zwei Instanzen des Typs RadiatorAct (wie in der Objektstruktur in Tabelle 6-2 definiert). In TempCtrlCtrl wird das Verhalten mit Hilfe eines SDL-Prozesses [OFM94, S.55ff.] definiert (siehe unten).

Neben der Aggregationsstruktur wird in diesen Diagrammen auch die Kommunikationsstruktur mit Hilfe von Kanälen [OFM94, S.118f.] festgelegt. Da für jede ausgehende und eingehende Verbindung von und zu TempCtrlCtrl ein eigener Kanal definiert wird, lassen sich die über diese Kanäle laufenden Signale durch Angabe der Signallisten der aggregierten Objekttypen beschreiben (siehe obere Hälfte von Abb. 6-6).

Das Verhalten von TempCtrl wird in einem SDL-Prozess TempCtrlCtrl auf der Basis von erweiterten Zustandsautomaten spezifiziert (siehe [OFM94, S.55ff.] und Abschnitt 6.2.3 auf Seite 151). In Abb. 6-7 ist zur Illustration die Transition von Task T7 in der SDL-Notation gezeigt.

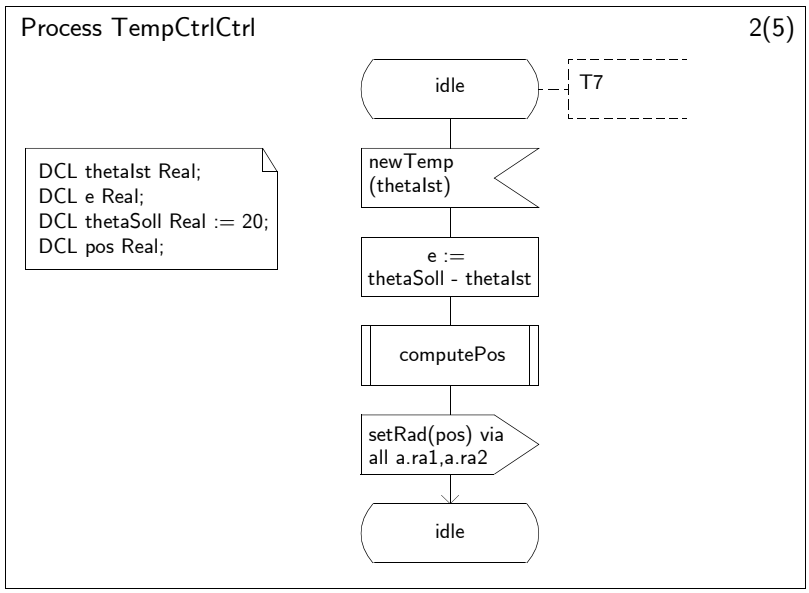


Abbildung 6-7. Beispielhafte Realisierung des Verhaltens in SDL

Die Bedeutung der hier verwendeten SDL-Symbole ergibt sich aus der Erläuterung der HTML-Syntaxelemente von oben. Eine präzisere Definition dieser Verhaltenselemente erfolgt in Abschnitt 6.2.3 auf Seite 151.

Aus allen SDL-Objekttypen wird schließlich ein ausführbarer Prototyp erzeugt. Dies erfolgt unter Verwendung einer generischen Modellbibliothek zur Verwaltung der Instanzen, zur Ein- & Ausgabe und für rudimentäre Berechnungen und String-Operationen (siehe auch Kapitel 9).

Das Ergebnis eines Laufs eines solchen Prototyps für unsere Temperaturregelung ist in Abb. 6-8 dargestellt.

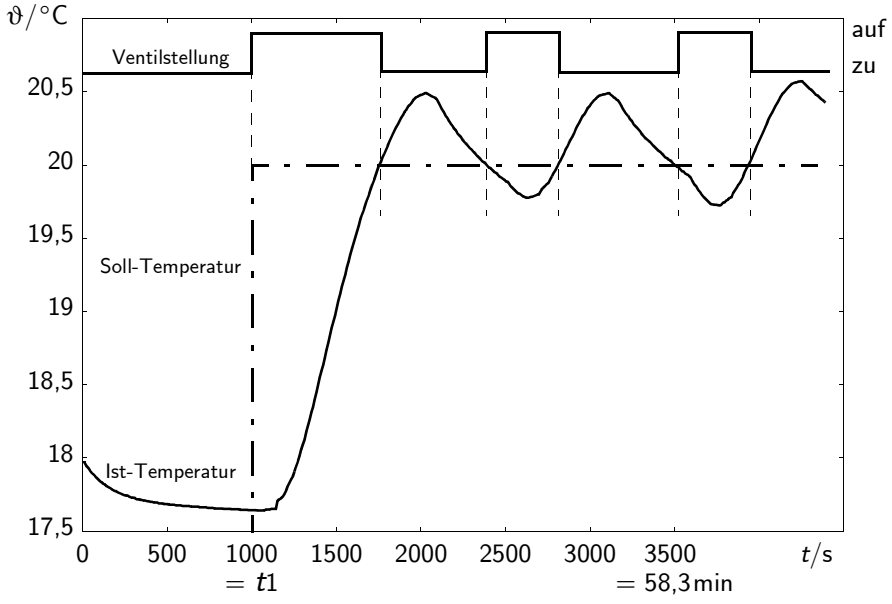


Abbildung 6-8. Gemessener Temperaturverlauf der Temperaturregelung inkl. Stellgröße

Dazu wurde das *RoomAutomation*-System an den von Riegel entwickelten Gebäudesimulator (siehe [Rie02] und [Rie04]) angekoppelt.

Initial besaß der Raum eine Temperatur von 18 Grad Celsius und die Solltemperatur lag bei 15 Grad. Erst ab dem Zeitpunkt $t_1 = 1000$s wurde die Solltemperatur auf 20° erhöht, womit die Regelung begann.

Neben dem Temperaturverlauf ist zusätzlich die Stellgröße (Heizkörperventil) angezeigt. Wie man erkennt zeigt die primitive Regelung deutliche Überschwinger (die Raumtemperatur schwankt um die Sollvorgabe von 20 Grad Celsius). Man könnte nun also eine Verbesserung dieses Regelalgorithmus vornehmen. Desweiteren könnte man daran gehen, die Energieeffizienz der Regelung zu optimieren, indem man die Belegung des Raumes berücksichtigt. In Abschnitt 9.3 auf Seite 275 werden wir mögliche Strategien zur Energieeinsparung experimentell untersuchen.

6.1.4 Verfolgbarkeit/Nachvollziehbarkeit

Wie man an Hand der obigen Beispiele gesehen hat, wird in der PROBAnD-Methode starkes Gewicht auf eine explizite *Verfolgbarkeit* (auch *Nachvollziehbarkeit*) zwischen Artefakten gelegt. So lässt sich an Hand der realizes-Spalte in der Task-Liste jederzeit die Abhängigkeiten zwischen Anforderungen ablesen und mit Verfolgbarkeitsrelationen bis hin zu einzelnen Objekttypen auch die Implementierung der Anforderungen nachvollziehen.

Insofern tragen diese Relationen zur *Post-Traceability* [Poh96] (auch „*post-RS traceability*“ [GoF94] oder *Entwurfsnachvollziehbarkeit* [VIS04]) bei. Die *Pre-Traceability* (auch „*pre-RS traceability*“ oder *Quellennachvollziehbarkeit*) erlaubt im Gegensatz dazu nachzuvollziehen, wie die Anforderungen entstanden sind.

Wie Gotel und Finkelstein in [GoF94] korrekt beobachten, verfügen die meisten Methoden zumindest über eine rudimentäre Unterstützung für eine solche Verfolgbarkeit. Dies kann in einer benutzerdefinierten Art und Weise erfolgen oder durch eine explizite Unterstützung in speziellen Arbeitsumgebungen (als Beispiel sei hier das Zusammenspiel der Werkzeuge Telelogic Doors und Tau genannt).

Obwohl einige der spezialisierteren Werkzeuge eine automatische Unterstützung für die Erstellung der Verfolgbarkeits-Links besitzen, muss die meiste Information von den Entwicklern vorgegeben werden (so auch in unserer PROBAnD-Methode). Daher sollten die Entwickler entsprechend motiviert werden, solche Links zu erzeugen und auch zu warten. Obwohl diese Informationen das endgültige Projekt nicht direkt voranzubringen scheinen und man daher sicherlich zum Teil ein Mangel an Motivation der Entwickler beobachten wird, glaube ich, dass sobald die Vorteile solcher Verfolgbarkeit bekannt sind, die Motivation entsprechend steigen wird. Auch der anfallende Aufwand zur Erstellung und Wartung der Verfolgbarkeitsinformation sollte nicht unterschätzt werden. Mögliche Vorteile sind jedoch z.B. die in Ab-

schnitt 8.2 auf Seite 208 vorgestellte automatische Detektion von Feature-Interaktionen oder die sehr einfache Identifikation derjenigen Stellen eines Produkts, die von einer Änderung der Anforderungen betroffen sind.

Wie bei jeder manuellen Tätigkeit besteht allerdings auch bei der Bearbeitung von Verfolgbarkeits-Links die Gefahr, dass sie fehlerhaft durchgeführt wird. Hier kann die Automatisierung anderer Aktivitäten allerdings helfen, solche Fehler aufzuspüren. Die erwähnte Feature-Interaktionsdetektion würde z.B. in solchen Fällen fehlerhafte Ergebnisse liefern, die eine weitergehende Untersuchung der Ursachen nach sich ziehen würde.

Die Verfolgbarkeit zwischen Artefakten ist ein sehr wichtiger Aspekt der PROBAnD-Methode, von der auch die Übertragbarkeit mancher der Ergebnisse dieser Arbeit abhängt, da viele der vorgestellten Automatisierungswerkzeuge die Information bzgl. der Verfolgbarkeit benötigen.

6.2 Das Produktmodell der PROBAnD-Methode

Nachdem wir im letzten Abschnitt eine Übersicht über die PROBAnD-Methode gaben, können wir nun deren Produktmodell detailliert vorstellen. Wie im letzten Kapitel bereits eingeführt wurde, ist dieses Produktmodell eine Instanz des dort vorgestellten Produktmetamodells.

Vorweg soll angemerkt werden, dass eine sehr wichtige Entwurfsstrategie bei der Spezifikation des Produktmodells von PROBAnD war, keine redundanten Relationen einzufügen, d.h. solche, deren Existenz auch aus anderen Relationen abgeleitet werden kann. Damit erspart man sich viele Konsistenzprüfungen auf der Ebene der abstrakten Artefakttypen und muss eine solche Prüfung nur auf den konkreten Artefakttypen durchführen (siehe dazu Abschnitt 8.1 auf Seite 188).

Desweiteren beziehen sich die in den folgenden Modellen angegebenen Multiplizitäten auf eine vollständige Spezifikation am Ende der Analysephase. *Während* der Entwicklung gilt für jedes Multiplizitäts-Constraint eine untere Grenze von „0“, da die Spezifikation noch nicht vollständig sein muss.

Im Folgenden werden wir das Produktmodell unterteilt in einzelne Kategorien vorstellen. Insbesondere die in Abschnitt 6.2.3 auf Seite 151 vorgestellten Artefakttypen stellen dabei eine Erweiterung der ursprünglich von Queins in [Que02] definierten PROBAnD-Methode dar. Nur für diese Artefakttypen werden wir die konkrete Syntax angeben, da für die anderen Artefakttypen dies bereits in [Que02] erfolgte, bzw. im *RoomAutomation*-Beispiel veranschaulicht wurde.

6.2.1 Beschreibung der Anforderungen und der Objektstruktur

Die atomaren Artefakte, die bei der Anwendung der PROBAnD-Methode zuerst spezifiziert werden sind solche, welche die Anforderungen und die Objektstruktur beschreiben. In Abb. 6-9 sind die Typen dieser atomaren abstrakten Artefakte gezeigt.

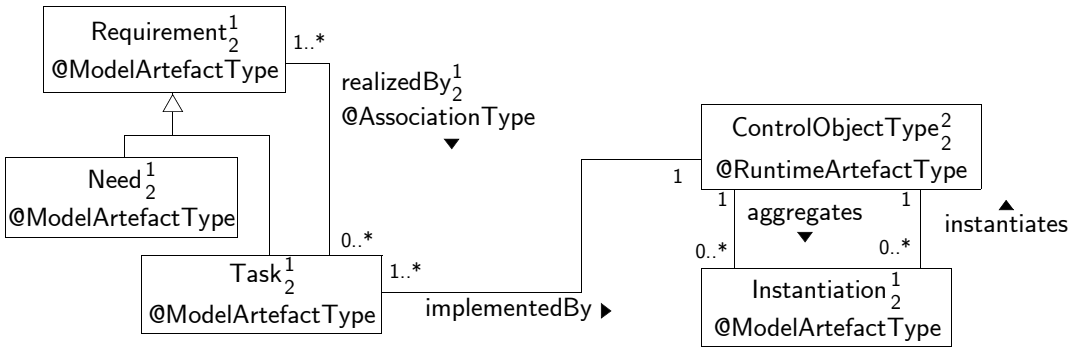


Abbildung 6-9. Strukturbeschreibende atomare Artefakttypen

Wie wir bereits zu Beginn dieses Kapitels eingeführt hatten, werden Needs durch Tasks realisiert, welche in eventuellen Sub-Tasks weiter verfeinert werden können. Sowohl Needs als auch Tasks sind Anforderungen und werden daher als Spezialisierung von Requirement beschrieben. Alle diese Anforderungstypen sind Instanzen von ModelArtefactType, da sie nur bis auf Ebene M_1 instanziiert werden (siehe Abschnitt 5.1.1 auf Seite 86).

Die Zuordnung der Tasks zu ControlObjectTypes erfolgt über Instanzen der implementedBy-Relation. Um die Aggregation von ControlObjectTypes zu einem Baum zu beschreiben, existiert der ModelArtefactType Instantiation. Anders als die ControlObjectTypes, welche als Instanzen von RuntimeArtefactType auf der Ebene M_0 existieren können (dieses sind letztlich die Objekte des ausgeführten reaktiven Systems), werden Instantiations nur auf Modellebene sichtbar. Das ist verständlich, wenn man bedenkt, dass sie nur beschreiben, wie die eigentlichen Instanzen erzeugt werden und danach nicht mehr benötigt werden.

Der obige Ausschnitt aus dem Produktmodell zeigt (wie auch alle folgenden) zur Wahrung der Lesbarkeit nicht immer die Typen der Relationen und keine Rollennamen an den Relationsenden. Diese Rollennamen lassen sich jedoch sehr leicht nach folgendem Schema ableiten: In Richtung des Pfeiles am Assoziationsnamen wird die Passiv-Form des Verbs mit dem Namen des Typs am Relationsende konkateniert. Für die aggregates-Assoziation ergibt sich so z.B. aggregatedInstantiation. In umgekehrter Richtung wird die Aktiv-Form zusammen mit dem entsprechenden Typnamen verwendet, also z.B. aggregatingControlObjectType. Sind die Assoziationsnamen bereits in der Passiv-Form (so z.B. bei der realizedBy-Relation) wird das Namensschema umgekehrt verwendet, also wie z.B. in realizedRequirement. Handelt es sich bei dem Verb des Assoziationsnamens um ein Hilfsverb (z.B. „is“ oder „has“),

dann wird its mit dem Typnamen konkateniert, also z.B. itsDataType (siehe weiter unten).

Die Dokumente, welche konkrete Artefakte obiger abstrakten Artefakte beinhalten sind die Liste der Needs (aus der Problembeschreibung), die Task-Liste und die Objektstruktur (siehe auch *RoomAutomation*-Beispiel in Abschnitt 6.1.3 auf Seite 138).

6.2.2 Statische Beschreibung von Objekteigenschaften und -kommunikation

Ein erster Schritt zu einer operationalen Spezifikation, die als Eingabe für das Prototyping dienen kann, ist die Beschreibung der für eine Verhaltensbeschreibung notwendigen statischen Objekteigenschaften und Kommunikationsbeziehungen (siehe Abb. 6-10).

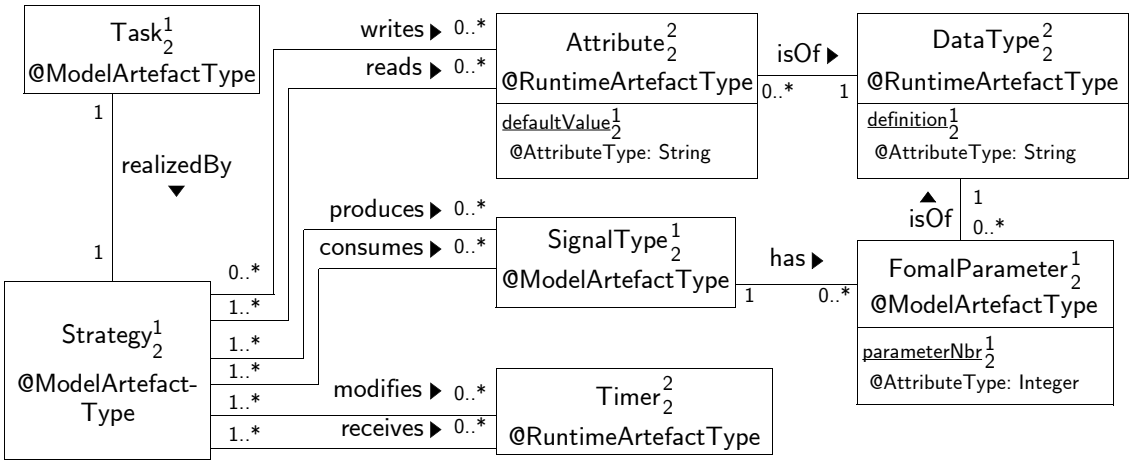


Abbildung 6-10. Artefakttypen zur Beschreibung des statischen Verhaltens

Beginnend mit einer natürlichsprachlichen Formulierung einer möglichen (funktionalen) Lösungsstrategie (Strategy) für einen gegebenen Task, werden die von einer solchen Strategy benötigten Attribute, SignalTypen und Timer beschrieben.

Die Semantik von Attribute und Timer (beide sind RuntimeArtefactTypes, da sie zur Laufzeit existieren können) entspricht dabei der Semantik von SDL (siehe z.B. [OFM94]), da dies der Formalismus des Modells, das beim Prototyping zur Ausführung gebracht wird, ist (siehe dazu Kapitel 9). Attribute weisen einen bestimmten Datentyp (DataType) auf, dessen Definition bei einem benutzerdefinierten Datentyp (z.B. einem Array) im Attribut definition abgelegt wird.

Mit den beiden Relationen von Strategy zu SignalType kann beschrieben werden, Signale welchen Typs von der Strategie erzeugt oder empfangen werden können. SignalType ist daher eine Instanz des ModelArtefactTypes. Das „zugehörige“ RuntimeArtefactType Signal wird in Abschnitt 6.2.3 eingeführt.

Da Signale in SDL Parameter besitzen können (siehe nächster Abschnitt), beinhaltet die Definition eines SignalTypes eine Liste eventueller formaler Parameter (FormalParameter), von denen jeder Parameter einen bestimmten Datentyp aufweist. Das Attribut parameterNbr dient der eindeutigen Kennzeichnung der Position des formalen Parameters in der Parameterliste. Dies ist notwendig, da wir an den Link-Enden stets von einer Menge von Instanzen ausgehen und damit keine Reihenfolge oder Ordnung vorgegeben wird (siehe dazu auch Abschnitt 4.2.2 auf Seite 65 und Abschnitt 5.2.1 auf Seite 97).

Dokumenttypen, welche die Artefakttypen aus Abb. 6-10 beinhalten, sind die HTML- und SDL-Dokumente zur Beschreibung der Control-Object-Types und die HTML-Dokumente, welche die Definition aller Signaltypen und Datentypen beinhalten (siehe dazu [Que02, S. 79 und 74]).

6.2.3 Operationale Verhaltensbeschreibung

Nach der Spezifikation der „statischen" Aspekte müssen nun auch die Elemente zur Verhaltensbeschreibung im Produktmodell spezifiziert werden. Dazu nutzen wir als zentrales Konzept das der *kommunizierenden erweiterten endlichen Automaten*, da dieses Automatenmodell auch der SDL-Semantik zu Grunde liegt [OFM94, S. 55ff.]. Im folgenden Abschnitt werden wir zunächst verschiedene Automatenmodelle vorstellen und damit das Konzept der erweiterten Endlichen Automaten einführen. Daran anschließend werden wir die Anpassung und Abbildung dieses Automatenmodells für die Artefakte der PROBAnD-Methode erläutern.

Allgemeine Automatenmodellierung

In Abschnitt 3.1.2 auf Seite 35 hatten wir die Endlichen Automaten bereits kurz eingeführt. Hier wollen wir den Automatenbegriff präzisieren und genauer definieren.

In der theoretischen Informatik wird ein *deterministischer Endlicher Automat*, also ein Automat mit deterministischem Verhalten, i.d.R. wie folgt definiert ([VIS04] und [Rem91, S. 98]):

> **Definition 6-1:** Ein **deterministischer Endlicher Automat** ist ein Sieben-Tupel $A = (I, O, Q, \delta, \omega, q_0, F)$, wobei I das Eingabealphabet (alle gültigen Eingabezeichen), O das Ausgabealphabet, Q eine endliche, nichtleere Menge von Zuständen, $q_0 \in Q$ der Startzustand und $F \subseteq Q$ die Menge der Endzustände ist. Die Zustandsübergangsfunktion (Transition) δ: $Q \times I \rightarrow Q$ ist eine Abbildung der Paare (q, i) mit $q \in Q$ (Ausgangszustand) und $i \in I$ auf einen Folgezustand $q' \in Q$.

Nach der Art der Ausgabefunktion ω werden weiter zwei Typen Endlicher Automaten unterschieden:

Definition 6-2: Ist die Ausgabefunktion ω: $Q \times I \rightarrow O$ von Eingabe und Zustand abhängig, so handelt es sich um einen **Mealy-Automaten**.

Definition 6-3: Ist die Ausgabefunktion ω: $Q \rightarrow O$ ausschließlich vom aktuellen Zustand abhängig, so spricht man von einem **Moore-Automaten**.

Diese obigen Definitionen lassen sich so nicht direkt im Kontext von objektorientierten Software-Artefakten verstehen. Dazu muss zunächst genauer geklärt werden, was die Eingabemenge I und die Ausgabemenge O beinhalten. In Abschnitt 3.1.2 auf Seite 35 hatten wir dies bereits erläutert. Die Eingabe in den Automaten sind die beobachtbaren Ereignisse („Events“). Die Ausgabe des Automaten sind Aktionen (siehe auch [RJB99, S. 439]). Wie bereits in Abschnitt 3.1.2 erläutert wurde, findet die Kommunikation zwischen Objekten mit Hilfe von Nachrichten statt. Daher beinhaltet I die Menge der Typen von empfangbaren Nachrichten und O die Menge der Nachrichtentypen, die von dem jeweiligen Objekt versendet werden können.

Typischerweise sind die Aktionen nicht anhaltend (so dauert das Versenden einer Nachricht nur eine sehr kurze Zeit [RJB99, S. 441]), weshalb wir im Folgenden stets das Mealy-Modell zu Grunde legen wollen.

Das obige Modell der Endlichen Automaten ist allerdings sehr einschränkend. Wollte man damit z. B. die Anzahl „empfangener“ Nachrichten zählen, müsste man eine große Zahl von Zuständen explizit modellieren. Um dieses Defizit auszugleichen führt man die erweiterten Endlichen Automaten (*Extended Finite State Machines*, *EFSM*s [OFM94, S. 55ff.]) ein, welche zusätzlich die Verwendung von Variablen zur Speicherung des Zustandes erlauben. Damit werden bedingte Transitionen mittels „Guards“ [RJB99, S. 291f.] und die Beschreibung *spontaner Zuständsübergänge* [OFM94, S. 74] auf Grund einer Änderung von Variablenbelegungen möglich. Desweiteren erfolgt eine Erweiterung um parameterbehaftete Nachrichten. Eine formale Definition der EFSMs findet man u. a. in dem Artikel von Boroday et al. [BGP03, S. 147].

Obiges Problem könnte man nun also durch die Definition einer Integer-Variablen lösen. Den Preis, den man für eine solche Erweiterung zahlt ist, dass es nicht mehr notwendigerweise möglich ist, die Eigenschaften des Automaten in endlicher Zeit zu prüfen (siehe [BrH93, S. 54]), insbesondere wenn man Datentypen mit unendlichen Wertebereichen zulässt.

Eine weitere Erweiterung der EFSMs sind die ursprünglich von Harel eingeführten *Statecharts*, die auch einen Teil der UML bilden (siehe z. B. [BHK04, S. 169ff.]). Diese erlauben zusätzlich eine Hierarchisierung (dekomponierte Zustände oder „OR-

States“) und eine Quasi-Parallelisierung (orthogonale Zustände oder „AND-States“). Wir wollen diese Statecharts allerdings nicht genauer untersuchen, sondern die in SDL eingesetzte Variante der EFSMs betrachten.

Obwohl die Verhaltensmodellierung in SDL auf dem Konzept der EFSM basiert, unterscheidet sich diese in einigen Punkten (siehe [BGP03]):

1. Transitionen in SDL korrespondieren nicht direkt mit den Transitionen der EFSMs [BGP03, S.149]. Dies liegt daran, dass in SDL Verzweigungen an beliebigen Stellen in einer Transition erlaubt sind und dadurch beliebige Aktionssequenzen realisiert werden können. Bei EFSMs kann man nur die Ausführung der *gesamten* Transition durch die Angabe eines Guards einschränken.
2. Anders als EFSM-Modelle besitzen die SDL-Modelle Kenntnisse über die Zeit. Der „Zugriff“ auf die Zeit wird durch den Operator now ermöglicht. Durch die Spezifikation von Timern (oder „Stoppuhren“, siehe [OFM94, S.102ff.]), können die Zeitaspekte eines Modells angegeben werden. Die Semantik von Timern und der Zeit in SDL ist komplex. Wir verweisen an dieser Stelle daher auf den Artikel von Prinz [Pri03], der dies genauer erläutert.
3. SDL erlaubt die Definition von Prozeduren, die aus einer SDL-Transition heraus aufgerufen werden können.
4. Anders als Zustandsautomaten besitzen SDL-Prozesse (deren Verhalten durch SDL-Transitionen beschrieben werden) eine (unendliche) Signalwarteschlange („Queue“). Signale (oder Nachrichten) gehen daher nur dann verloren, wenn sie im aktuellen Zustand bzw. im Folgezustand der gerade abgearbeiteten Transition nicht entgegengenommen und nicht durch ein sog. save-Konstrukt „gespeichert“ werden (siehe [OFM94, S.78ff.]).

Die Kenntnis dieser Punkte ist für das Verständnis unserer Verhaltensartefakte notwendig, da wir uns hierbei an der SDL-Semantik orientieren.

Modifizierte Automatenmodellierung

Ein Nachteil aller obigen Automatenmodelle ist, dass stets der Automat als Ganzes betrachtet wird. Das kann Probleme bei der Skalierung bedeuten, aber auch eine feingranulare Verfolgbarkeit von Anforderungen zu den relevanten Teilen des Automaten verbieten. Eine Hierarchisierung von Automaten (wie z.B. bei den Statecharts) stellt keine echte Lösung dieses Problems dar, weil in vielen Fällen keine eindeutige Zuordnung von Anforderungen zu hierarchischen Zuständen möglich ist. Es existieren oft Überschneidungen, da verschiedene Tasks Abhängigkeiten bzgl. einer Transition aufweisen können. Aber auch die Verteilung von abhängigen Transitionen eines einzelnen Tasks über den gesamten Automaten ist möglich.

Der Kerngedanke bei unserer modifizierten Automatenmodellierung ist daher, jede Strategie durch eine oder mehrere Zustandsübergänge zu beschreiben. Das Ge-

samtverhalten eines Control-Object-Types ergibt sich dann aus der Menge aller Transitionen der von einem Objekttyp implementierten Tasks (bzw. deren Strategien). Man beschreibt also „Ausschnitte“ aus einem großen (und unter Umständen sehr komplexen) Zustandsgraphen. In Abb. 6-11 ist dieses Konzept an einem einfachen Beispiel veranschaulicht.

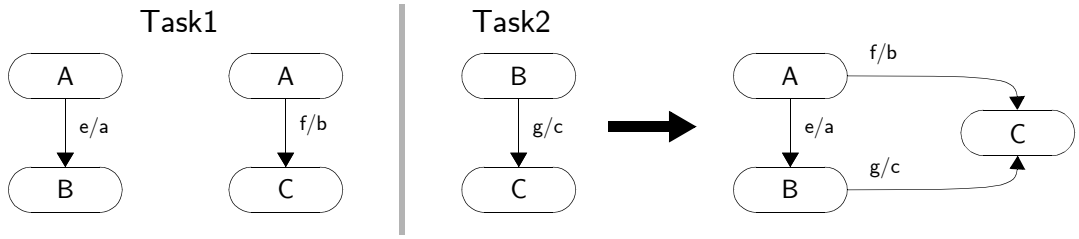


Abbildung 6-11. Komposition einzelner Transitionen zu einem Gesamtautomaten

Bei der Erzeugung des Gesamtgraphen aus den Teilgraphen kann es natürlich zu Konflikten kommen (z.B. führen unterschiedliche Zielzustände bei gleichem Anfangszustand und gleichem Ereignis, das die Transition auslöst, zu einem nichtdeterministischen Automaten), die man auflösen muss. In unserem Ansatz obliegt diese Aufgabe dem Modellierer, der auf solche Konflikte bei der Analyse der Modelle hingewiesen wird (siehe dazu Abschnitt 8.1 auf Seite 188).

Abb. 6-12 zeigt mit welchen atomaren Artefakttypen die einzelnen Transitionen beschrieben werden.

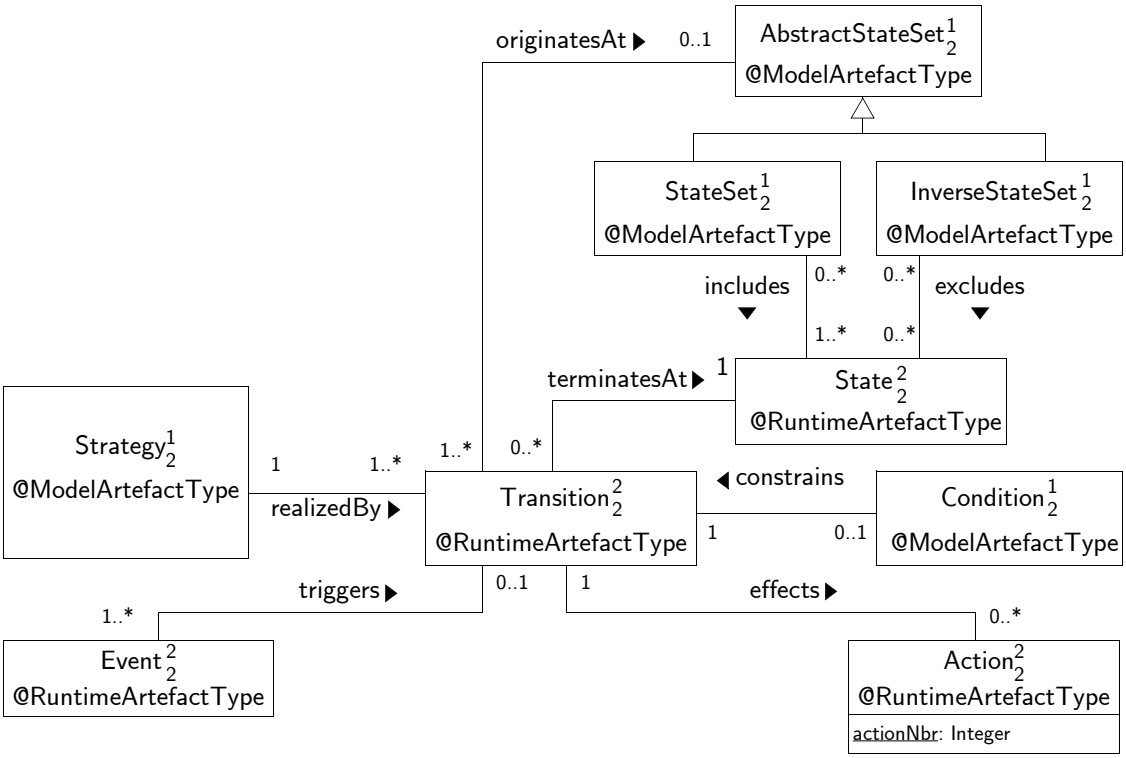


Abbildung 6-12. Artefakttypen zur Beschreibung des dynamischen Verhaltens

Jede Transition kann einen Link (originatesAt) zu einer Instanz des Artefakttyps AbstractStateSet aufweisen, welcher einer Menge von Ausgangszuständen beschreibt. Ist die originatesAt-Assoziation nicht belegt – und damit kein Ausgangszustand angegeben – dann wird die Transition als Initialisierungstransition interpretiert. Diese wird bei der Instanziierung des zugehörigen Control-Object-Types automatisch ausgeführt.

Die Beschreibung der Ausgangszustände durch AbstractStateSet erlaubt die Aufzählung der Zustände auf zwei Arten. Durch den spezialisierten Artefakttyp StateSet erfolgt die explizite Angabe einer Menge existierender Zuständen (State), die als Ausgangszustände dienen (includes-Assoziation). Beinhaltet die Menge nur einen Zustand, dann handelt es sich um den typischen Ausgangszustand eines Endlichen Automaten. Mit Hilfe des Artefakttyps InverseStateSet kann spezifiziert werden, dass alle Zustände außer die über die excludes-Relation angegebenen als Ausgangszustand akzeptiert werden. Mittels des InverseStateSet kann somit der „Asterisk-State“ („*“) von SDL beschrieben werden (siehe [OFM94, S.37]).

Neben einem oder mehrerer Ausgangszustände besitzt jede Transition exakt einen Folgezustand, der über die terminatesAt-Relation spezifiziert wird.

Typischerweise wird jede Transition von einem Ereignis (Event) ausgelöst. Existiert keine Angabe eines Events über die triggers-Assoziation, dann handelt es sich um eine spontane Transition, welche zu einem beliebigen Zeitpunkt auftreten kann.

Bei Auftreten des spezifizierten Ereignisses wird die Transition ausgelöst und eine beliebige Anzahl von Aktionen (effects-Assoziation) ausgeführt. Dabei gibt das Attribut actionNbr die Reihenfolge der einzelnen Aktionen innerhalb einer Transition an. Das zu Grunde liegende Automatenmodell ist, wie oben bereits erwähnt, das der Mealy-Automaten (siehe Definition 6-2 auf Seite 152).

Die Ausführung von Aktivitäten lässt sich nun weiter durch eine Bedingung (Condition) einschränken. Anders als bei Endlichen Automaten (oder Statecharts), wo ein Guard angegeben wird, der das Auslösen der Transition bei Nichterfülltsein der Guard-Bedingung verhindert, wird die Condition erst nach dem „Konsumieren“ des Ereignisses ausgewertet. Nur die Ausführung der Aktionen ist also von der Wahrheit der Bedingung abhängig.

Wir wählten eine solche Spezifikation von Bedingungen, da zum einen nicht in allen Fällen eine Abbildung der „Statechart“-Guards auf die „Continuous Signals“ bzw. „Enabling Conditions“ von SDL (vgl. [OFM94, S.75ff.]) möglich ist und zum anderen in den von uns betrachteten Systemen viele der Zustandsübergänge von den aktuellen Signalparametern abhängen (siehe Temperaturregelungsbeispiel aus Abschnitt 6.1.3 auf Seite 138). In letzterem Fall muss das Signal folglicherweise *vor* der Auswertung der Bedingung empfangen worden sein, da ansonsten die aktuelle Parameterbelegung nicht bekannt ist.

Die Realisierung dieser Conditions erfolgt in SDL über das Syntax-Konstrukt der „Decision“ [OFM94, S. 84ff.]. Unser Produktmodell schränkt die Verwendung dieses Konstrukts ggü. der SDL-Syntax jedoch ein. Wo in SDL beliebige Kontrollflüsse (und damit auch Graphen) möglich sind (vgl. Punkt 1 im vorhergehenden Abschnitt), ist im Produktmodell der PROBAnD-Methode nur eine Kaskade (Baum) von bedingten Verzweigungen beschreibbar. Besonders für die frühe Spezifikation von Verhalten (in den HTML-Dokumenten) erachteten wir die „programmiersprachenähnliche“ Mächtigkeit von SDL als nicht angebracht.

Die konkreten Artefakttypen (Views, siehe Abschnitt 5.1.1 auf Seite 86) zu obigen abstrakten Typen existieren sowohl für unsere HTML- als für die SDL-Notation. In Abb. 6-13 ist zur Veranschaulichung dieser Views ein Beispiel gezeigt, das zunächst von der weiter unten folgenden Verfeinerung der Events und Actions abstrahiert.

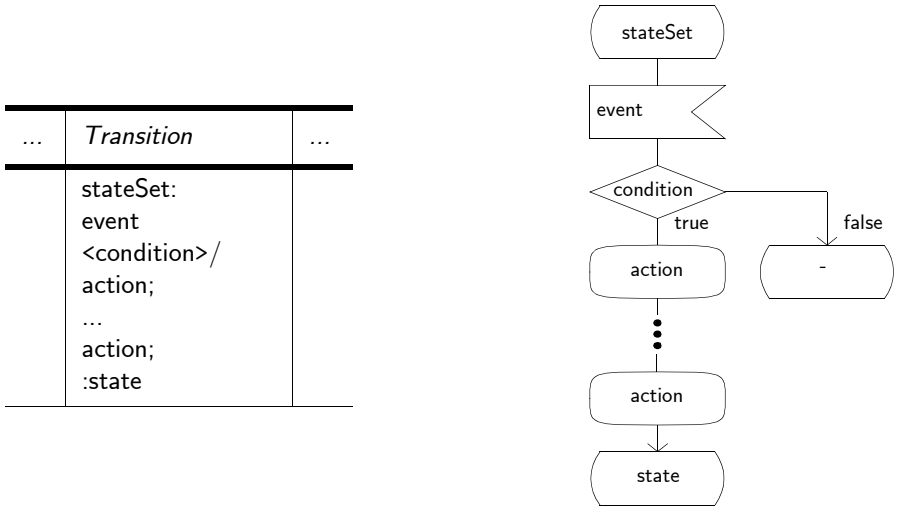


Abbildung 6-13. HTML- und SDL-Views der Artefakte aus Abb. 6-12

Die optionale Condition kann ein beliebiger SDL-Ausdruck sein. Werden mehrere Transitionen mit identischen Ausgangszustandsmengen und Ereignissen angegeben, so müssen diese durch die Angabe unterschiedlicher Conditions voneinander unterscheidbar sein (eine Vereinigung identischer Transitionen zu einer einzigen lassen wir wegen der dann nicht mehr möglichen Verfolgbarkeit nicht zu). Diese so durch Conditions konkretisierten Transitionen werden dann in der Reihenfolge ihrer Spezifikation kaskadiert. Dem Modellierer obliegt allerdings die Überprüfung, ob die gewählten Conditions in einer solchen Kaskade sinnvoll und die Aktivitäten wirklich erreichbar sind. (Auch die Telelogic Tau SDL Suite [LEH00] stellt keine statischen Analysen zur Überprüfung solcher Fehler zur Verfügung.)

Ereignis- und Aktionstypen

Die in Abb. 6-12 beschriebenen Artefakttypen Event und Action können zu weiteren, spezielleren Typen klassifiziert werden. In Abb. 6-14 sind alle solche Typen von Ereignissen und Aktionen gezeigt.

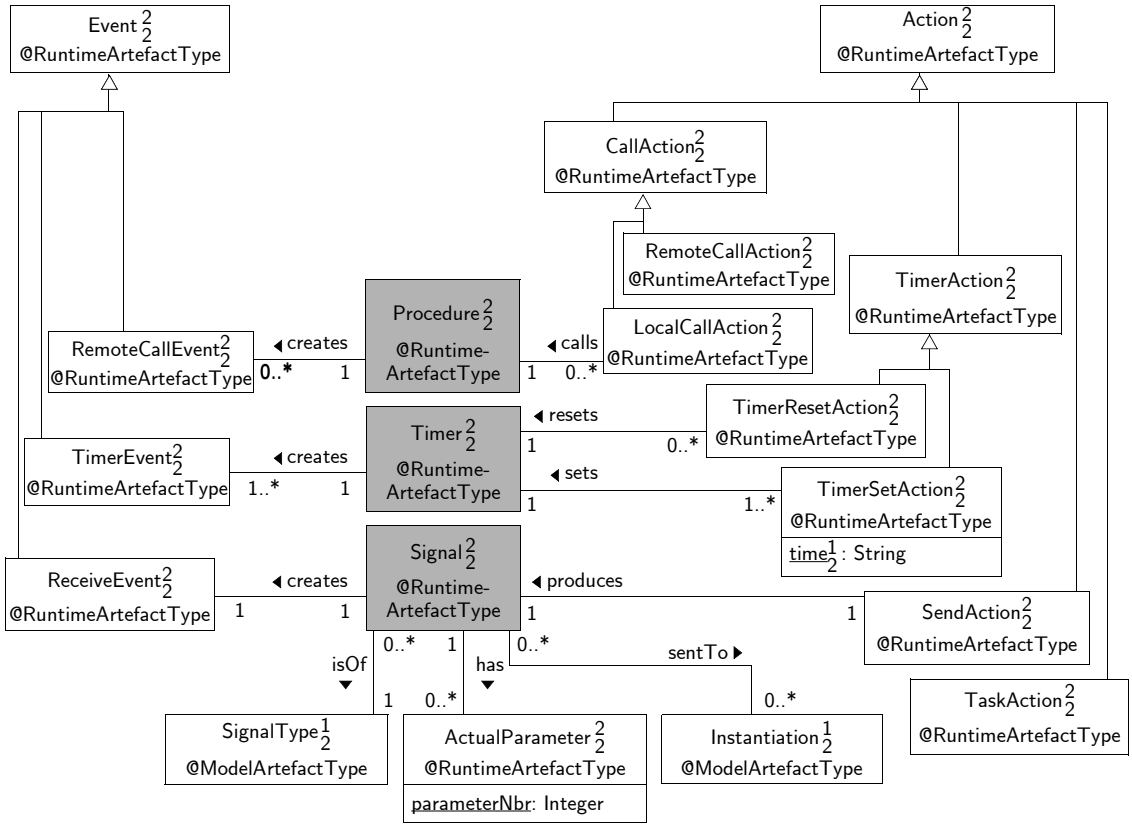


Abbildung 6-14. Spezielle Artefakttypen zur Beschreibung von Events und Actions

Die Typen von Ereignissen und Aktionen lassen sich abhängig von den sie bedingenden Objekten (in Abb. 6-14 grau hinterlegt) in drei Gruppen einordnen: Signal-, Timer- und Prozedurereignisse und -aktionen.

Ein typisches Ereignis ist der Empfang eines Signals (ReceiveEvent), welches durch das Versenden eines Signals desselben Typs (SendAction) ausgelöst wird. Beim Versenden eines Signals (SendAction) kann zusätzlich das Ziel des Signals angeben werden. Neben dem Namen von Instantiations (siehe Abb. 6-10 auf Seite 150) ist hier auch die Angabe von super erlaubt, was ein Senden der Signale zu dem aggregierenden Control-Object-Type bewirkt.

Da ein SignalType (siehe Abschnitt 6.2.2) eine Liste formaler Parameter besitzt, erhalten die Signale demzufolge eine Liste aktueller Parameter (has-Assoziation zu ActualParameter). Abhängig davon, ob ein Signal empfangen oder versendet wird können diese aktuellen Parameter unterschiedliche Belegungen annehmen. Bei dem Empfang eines Signals sind nur Variablennamen als aktuelle Parameter erlaubt, da

diese die konkreten Belegungen der Parameter aufnehmen müssen. Beim Versand eines Signals ist neben einem Variablennamen auch ein Literal des entsprechenden Datentyps oder sogar ein SDL-Ausdruck (zur Berechnung eines Wertes) erlaubt.

Der Artefakttyp Signal beschreibt also eine konkret empfangbare Signalinstanz und ist deshalb eine Instanz des Meta-Metatyps RuntimeArtefactType. Zur Veranschaulichung der Zusammenhänge zwischen SignalType, Signal und der zur Laufzeit existierenden Signalinstanz ist in Abb. 6-15 ein Beispiel für diese verschiedenen Artefakte gezeigt.

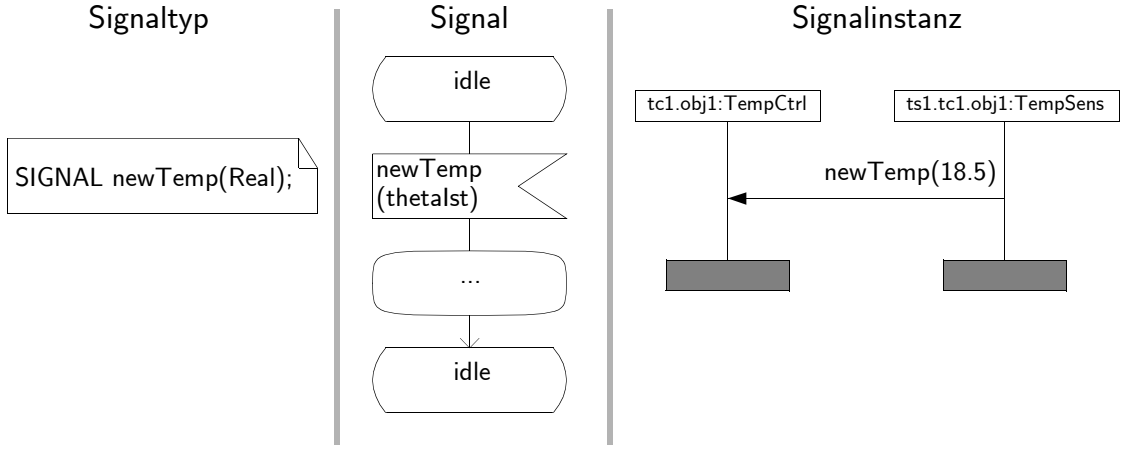


Abbildung 6-15. Signaltypen, Signale und Signalinstanzen am Beispiel

Weitere Ereignisse sind die TimerEvents, welche beim Ablauf eines Timers („Timeout“) ausgelöst werden. Entsprechende Aktionen sind das „Aufziehen“ (TimerSetAction) und das „Zurücksetzen“ (Stornieren) eines Timers (TimerResetAction). Das „Aufziehen“ eines Timers kann entweder unter Angabe eines absoluten Zeitwerts oder relativ zum aktuellen Zeitpunkt (gekennzeichnet durch ein „+“ zu Beginn der String-Belegung des time-Attributs) erfolgen.

Neben der Kommunikation mittels Signalen erlaubt SDL durch *Remote-Procedure-Calls* (*RPC*s, [OFM94, S.94ff.]) auch die Kommunikation von Prozessen ohne dass diese über einen Kanal verbunden sind. Dieser Mechanismus wird in der PROBAnD-Methode zur Realisierung der Kommunikationsbeziehungen der Sensoren und Aktuatoren zur technischen Schnittstelle zur Umgebung eingesetzt (siehe dazu [BrS03] und [Que02, S.193ff.]). Daher wird an dieser Stelle auch die Berücksichtigung dieses Konzeptes in den Produktmodellen notwendig.

Jeder Aufruf einer im betrachteten Control-Object-Type definierten Remote-Procedure (Procedure) löst ein beobachtbares Ereignis (RemoteCallEvent) aus. Genauso kann man in einer Aktion (RemoteCallAction) eine in einem entfernten Objekttyp definierte RPC aufrufen.

Will man Verhaltenskonstrukte beschreiben, die über das hinausgehen was das Produktmodell erlaubt (also z.B. Berechnungen mit Schleifen oder mit Case-Kon-

strukten), dann kann eine SDL-Prozedur definiert werden, die man durch die LocalCallAction bei Bedarf aufrufen kann. Anders als die RPCs sind die Prozeduren nur innerhalb des definierenden Control-Object-Types sichtbar (siehe dazu [OFM94, S. 89ff.]).

Da man häufig nur kurze Ausdrücke in einer solchen Prozedur definieren würde, wurde der Artefakttyp TaskAction eingeführt, welcher einen in SDL-Syntax angegebenen (kurzen) Ausdruck (z. B. dCon := thetaSoll - thetaIst) beinhaltet.

Die Notation der konkreten Artefakte zur Verhaltensbeschreibung soll anders als die der allgemeineren PROBAnD-Dokumente präzise definiert werden, da dieser Teil die bereits erwähnte Erweiterung der PROBAnD-Methode darstellt.

Konkrete HTML-Syntax für Transitionen

Zur Präzisierung der HTML-Views nutzen wir eine lineare Grammatik in BNF-Notation. Die HTML-Syntax zur Beschreibung einer Transition ist im Folgenden gegeben (`[element]` gibt optionale Elemente an, `|` die Alternative und `<element>` ein Nichtterminalsymbol).

```
1   <transition>          ::= <regularTransition> |
2                             <initTransition>
3
4   <regularTransition>   ::= <originatingStateSet> :
5                             [<triggeringEvent>]
6                             <transitionBody>
7                             : <terminatingState>
8
9   <initTransition>      ::= :
10                            <transitionBody>
11                            : <terminatingState>
12
13  <transitionBody>      ::= [< <constrainingCondition> >] /
14                            [<effectedActionList>]
15
16  <constrainingCondition> ::= "<expression>"
17
18  <effectedActionList> ::= [<effectedActionList> ;] <effectedAction>
19
20  <originatingStateSet>::= <stateSet> | <inverseStateSet>
21  <stateSet>            ::= [<stateSet>], <identifier>
22  <inverseStateSet>     ::= *(<stateSet>)
23  <terminatingState>    ::= - | <identifier>
```

Es werden zwei Arten von Transitionen in dieser Grammatik unterschieden, die normalen Transitionen (`regularTransition`) und solche, die zur Initialisierung dienen (`initTransition`). Eine `initTransition` besitzt anders als eine `regularTransition` keinen Ausgangszustand.

Als Ausgangszustand einer regulären Transition kann wie bei den abstrakten Artefakttypen des Produktmodells ein `StateSet` angegeben werden. Dieser besteht entweder aus einer mit Komma getrennten Liste von Zuständen oder er beginnt mit dem „`*`“ gefolgt von einer Liste ausgenommener Zustände in Klammern. Zustände werden durch einen eindeutigen Namen (`identifier`) gekennzeichnet. Als Folgezustand ist wie in SDL auch die Angabe von „`-`“ erlaubt, wodurch spezifiziert wird, dass der Folgezustand derselbe wie der Ausgangszustand ist (siehe [OFM94, S.37]).

Beide Transitionen besitzen einen Rumpf, den `transitionBody` (Zeile 13), welcher aus einer optionalen Bedingung (Condition), gefolgt von einer Liste von Aktionen bestehen kann.

```
24  <triggeringEvent>     ::= <receiveEvent> | <timerEvent> |
25                            <remoteCallEvent>
26
27  <receiveEvent>        ::= ^<identifier>[(<expressionList>)]
28  <timerEvent>          ::= @<identifier>
29  <remoteCallEvent>     ::= ?<identifier>
```

Als Ereignisse (`triggeringEvent`) sind der `receiveEvent` (beginnend mit „`^`“ und einer optionalen Liste von Parametern), der `timerEvent` (beginnend mit „`@`“) und ein `remoteCallEvent` (beginnend mit „`?`“) möglich.

```
30  <effectedAction>      ::= <sendAction> | <timerAction> | <callAction> |
31                            <taskAction>
32
33  <sendAction>          ::= ^<identifier>[(<expressionList>)]
34                            [-> <destination>]
35  <destination>         ::= super | <identifierList>
36  <timerAction>         ::= <absoluteSetAction> |
37                            <relativeSetAction> |
38                            <timerResetAction>
39  <absoluteSetAction>   ::= @<identifier>(<value>)
40  <relativeSetAction>   ::= +<identifier>(<value>)
41  <timerResetAction>    ::= -<identifier>
42
43  <callAction>          ::= <localCallAction> | <remoteCallAction>
44  <localCallAction>     ::= !<identifier>
45  <remoteCallAction>    ::= ?<identifier>[(<expressionList>)]
46
```

```
47  <taskAction>          ::= "<expression>"
```

Die Aktionen entsprechen den in Abb. 6-14 eingeführten Typen. Die `sendAction` beginnt mit dem Zeichen „^“ gefolgt von einer optionalen Liste von Signalparametern und einem optionalen Ziel (`destination`).

Bei den `timerActions` werden die `absoluteSetAction` (beginnend mit „@“), die `relativeSetAction` (beginnend mit „+“) und die `timerResetAction` (gekennzeichnet durch „-“) unterschieden.

Die beiden Arten der `callActions` werden durch „!“ für eine `localCallAction` und „?“ für eine `remoteCallAction` unterschieden.

Zuletzt kann – in Anführungszeichen eingeschlossen – eine `taskAction` spezifiziert werden.

```
48  <expressionList>      ::= [<expressionList>, ]<expression>
49  <expression>          ::= /* beliebige SDL Expression */
50
51  <identifierList>      ::= [<identifierList>, ]<identifier>
52  <identifier>          ::= /* beliebiger SDL Identifier */
```

In den letzten Zeilen der Grammatik werden die allgemeineren Nichtterminale (Listen von Ausdrücken und Identifiern) definiert. Für `expression` und `identifier` erfolgt hier ein Verweis auf die SDL-Grammatik (siehe z.B. [OFM94, S.341ff.]).

Konkrete SDL-Syntax für Transitionen

Die formale Spezifikation der SDL-Views erfolgt mit Graphgrammatiken (siehe Abschnitt 2.1.2 auf Seite 13). Werden innerhalb eines grafischen Symbols Ausdrücke in der Form <name> verwendet, so bezieht sich dies auf das Nichtterminal obiger linearer Grammatik.

Anders als bei den HTML-Views, wo eine Verfolgbarkeit der Transition zu dem zugehörigen Task (bzw. der zugehörigen Strategie) über die Zeile in der Tabelle hergestellt wird, müssen wir in SDL eine Annotation der jeweiligen Transition nach einem festgelegten Schema vornehmen. Dazu setzen wir bei unserer Lösung das Syntaxkonstrukt des SDL-Kommentars ein, mit dessen Hilfe wir bei dem Ausgangszustand einer Transition den Namen des realisierten Tasks angeben. Bei einer Kaskade von mit Conditions eingeschränkten Transitionen, wird eine Liste aller Tasks in der Reihenfolge der Transitionen spezifiziert. Eine solche Annotation ist in Abb. 6-16 für die abstrakte Darstellung aus Abb. 6-13 auf Seite 156 gezeigt.

In den Abbildungen 6-17 und 6-18 ist der Ausschnitt aus der Graph-Grammatik für den Transitionsrumpf gezeigt. Wie bei der linearen Grammatik wird auch hier zwischen einer regulären Transition und einer Initialisierungstransition unterschie-

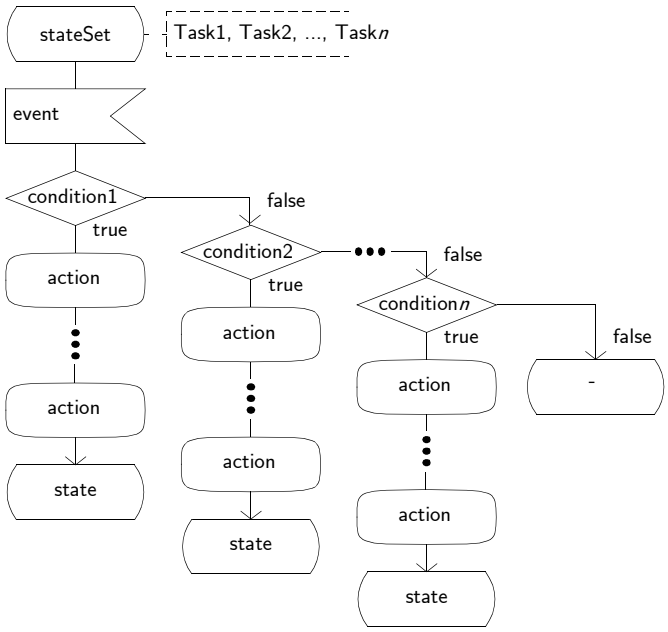


Abbildung 6-16. Annotation der SDL-Views zur Verfolgbarkeit von Anforderungen

den. Eine Feinheit ist allerdings bei der initTransition zu erkennen. An Stelle des Start-Symbols von SDL benutzen wir ein Join-Symbol, das die Verbindung zu einem vorhergehenden Teil der Transition herstellt. Dies ist notwendig, da wir für jeden Control-Object-Type einige grundlegende Verhaltenskonstrukte erzeugen (wie z.B. zur Berechnung der eindeutigen Instanzennamen). Da diese für den Modellierer verborgen bleiben können bzw. bei unserem automatisierten Ansatz automatisch regeneriert werden, besitzt man mit dem Join-Symbol einen standardisierten Haken zur Verbindung mit dem generierten Verhaltensteil.

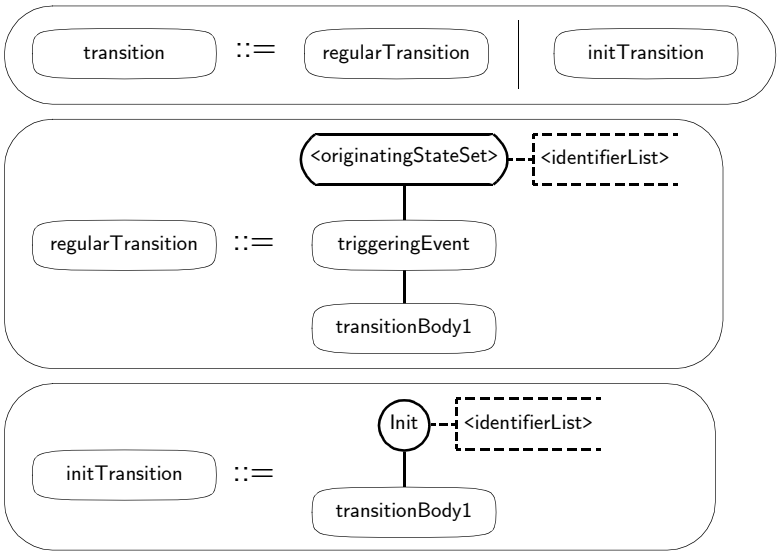


Abbildung 6-17. Graph-Grammatik für Transitionskörper (SDL) – Teil 1

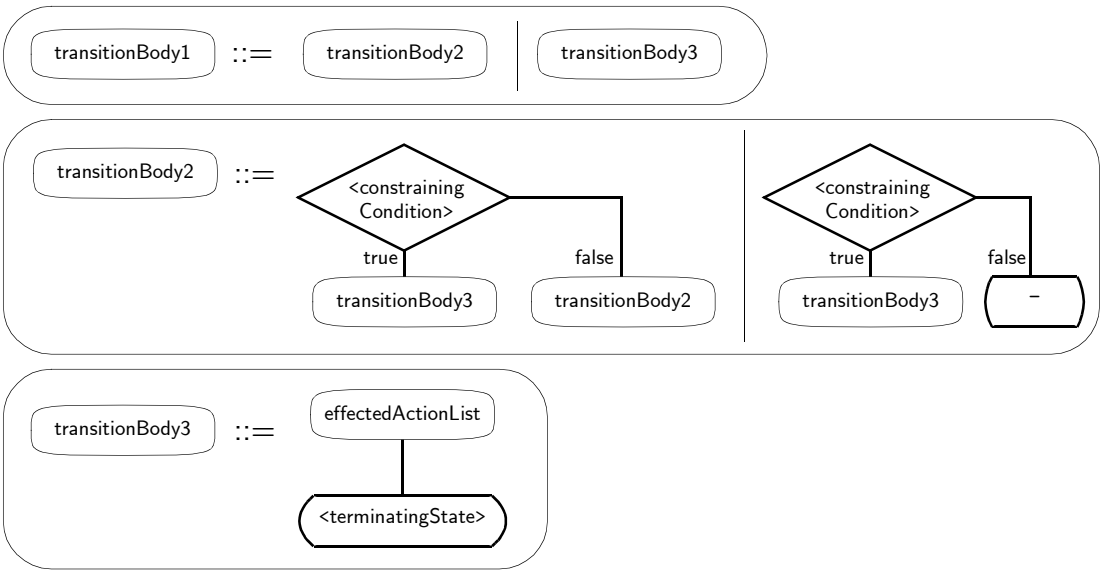


Abbildung 6-18. Graph-Grammatik für Transitionskörper (SDL) – Teil 2

Die restlichen Nichtterminale beschreiben die mögliche Kaskadierung von Transitionen durch Angabe von Conditions und die Liste von Aktionen für den Transitionsrumpf (transitionBody1 bis transitionBody3).

Die Ereignisse werden auf die entsprechenden grafischen Syntaxkonstrukte von SDL (wie z.B. das Input-Symbol für das receiveEvent) abgebildet (Abb. 6-19).

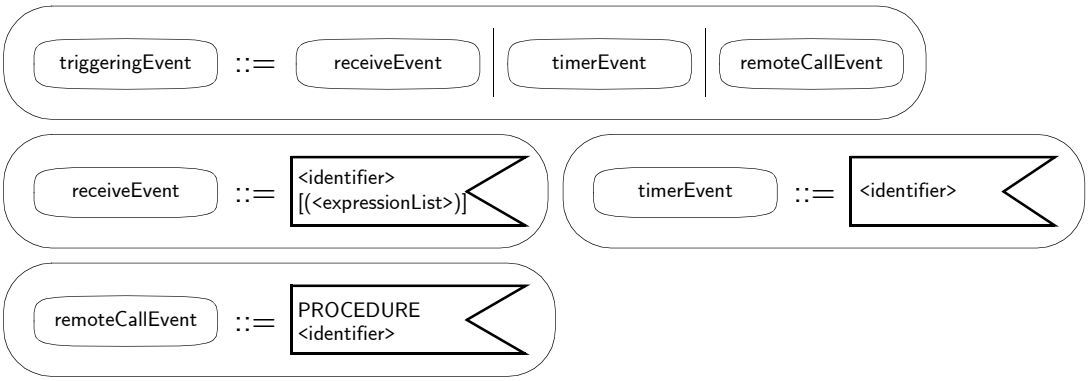


Abbildung 6-19. Graph-Grammatik für Ereignisse (SDL)

In Abb. 6-20 folgt die Syntax für die Aktionsliste (effectedActionList) und die ausgelösten Aktionen.

Auch hier werden wieder dieselben Aktionen wie in der linearen Grammatik unterschieden. Die sendAction wird auf das Output-Symbol von SDL abgebildet, wobei auch hier eine optionale Liste von Parametern angegeben und das Ziel des Signals spezifiziert werden kann. Dies erfolgt durch Angabe des Kanals, über (via) welches das Signal laufen soll. Da die Möglichkeit besteht mehr als einen Empfänger anzugeben ist hier eine Liste von Identifiern erlaubt. Zusätzlich spezifizieren wir mit dem Schlüsselwort all, dass das Signal an alle über die angegeben Kanäle erreichbaren

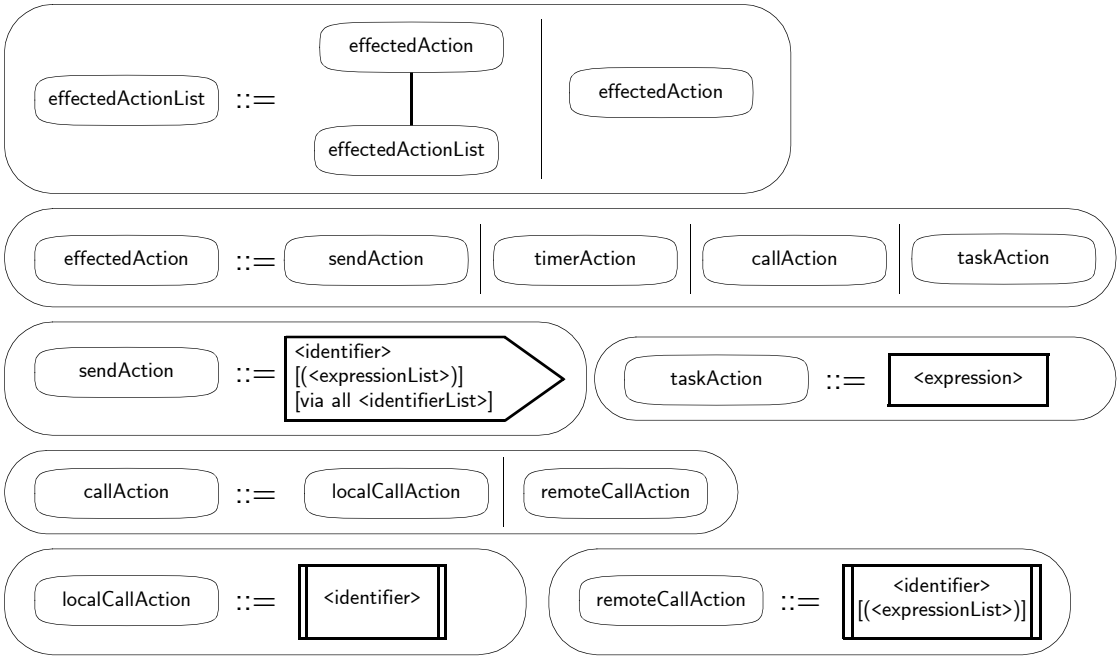


Abbildung 6-20. Graph-Grammatik für Aktionen (SDL), Teil 1

Instanzen gesendet wird. Ansonsten würde der Empfänger des Signals „zufällig“ aus der Liste der möglichen Empfänger ausgewählt (siehe [OFM94, S. 72f.]).

Die taskAction und die callActions werden wie erwartet auf SDL abgebildet. Es sei hier allerdings angemerkt, dass eine localCallAction (und demzufolge eine lokale Prozedur) keine Aufruf- und Rückgabeparameter besitzen darf. Die Prozeduren wurden in unserem Ansatz zur „Ausblendung“ mächtigerer Syntaxkonstrukte eingeführt (siehe oben) und dienen nicht primär der Definition wiederverwendbarer „Funktionen“.

Zuletzt geben wir in Abb. 6-21 die SDL-Syntax der timerActions an. Das Aufziehen eines Timers in SDL erfolgt mit dem Operator set, der einen absoluten Zeitpunkt (vom Datentyp Time) erwartet. Die relativeSetAction wird daher durch eine

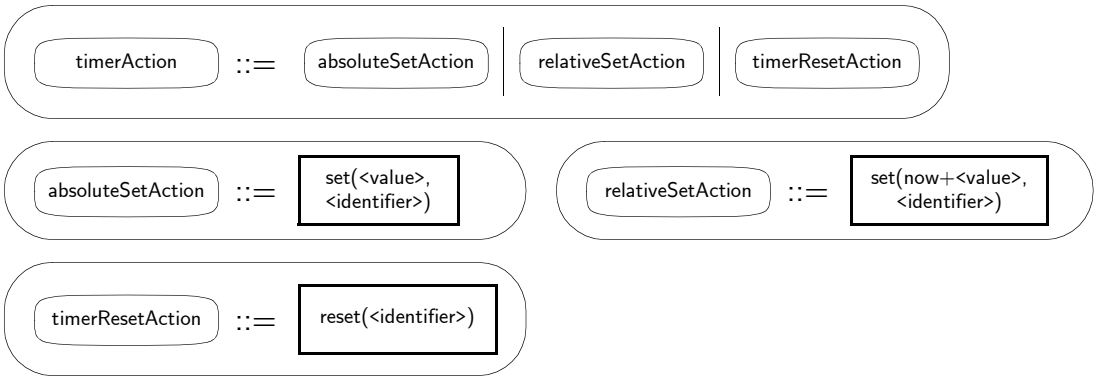


Abbildung 6-21. Graph-Grammatik für Aktionen (SDL), Teil 2

Addition des Zeitabstands <value> zur aktuellen Zeit (now) realisiert. Eine Stornierung (Zurücksetzen) eines Timers erfolgt mit dem Operator reset.

Zur Veranschaulichung der in diesem Abschnitt definierten Views verweisen wir auf Abschnitt 6.1.3, in welchem diese am *RoomAutomation*-Beispiel bereits vorgestellt wurden.

6.2.4 Komplexe Artefakttypen

Neben den oben vorgestellten abstrakten atomaren Artefakttypen und deren konkreten Repräsentationen in den Views (für die Verhaltenkonstrukte wurde im letzten Abschnitt deren Syntax angegeben) existieren auch komplexe Artefakttypen. Auf die konkreten Vertreter, die Dokumente, hatten wir bereits in den obigen Abschnitten jeweils hingewiesen. Die abstrakten komplexen Artefakttypen (Instanzen des Meta-Metatyps ComplexArtefactType) werden in diesem Abschnitt aufgeführt, da diese – wie in Abschnitt 5.1 auf Seite 82 eingeführt wurde – die für eine Modellmodifikation notwendigen Artefakttypen aggregieren.

Die ComplexArtefactTypes weisen wegen dieser Aggregationsbeziehung eine sehr einfache Struktur auf. Wir werden die für die PROBAnD-Methode definierten ComplexArtefactTypes daher in einer kompakten Tabellenform an Stelle der bisher verwendeten UML-Klassendiagramme notieren.

Beginnen werden wir mit dem ersten Artefakttyp, der in der Liste der Needs seine konkrete Entsprechung findet. Abb. 6-22 zeigt das Objektmodell und die entsprechende Information in Tabellenform.

Multiplizitaet	*aggregierter Artefakttyp*
0..*	Need

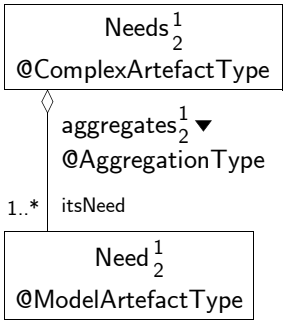


Abbildung 6-22. Komplexer Artefakttyp Needs

Neben der einfachen Darstellung in der Tabelle ist auch das Namensschema für die Benennung der Rollennamen erkennbar. Auf der Seite des aggregierten Artefakttyps ist dies immer its<NameDesArtefakttyps>. Deshalb ist es auch nicht notwendig die Rollennamen in der Tabelle zu wiederholen.

In Tabelle 6-9 ist der komplexe Artefakttyp der Task-Liste gezeigt. Neben dem offensichtlichen Artefakttyp Task werden von TaskList auch die Typen Need und ControlObjectType aggregiert, da ein Task zu beiden Typen eine Relation aufweist,

die auch in dem entsprechenden Dokument der Task-Liste (siehe z.B. Tabelle 6-3 auf Seite 140) aufgeführt wird.

Tabelle 6-9. Komplexer Artefakttyp TaskList

Multiplizitaet	*aggregierter Artefakttyp*
0..*	Task
0..*	Need
0..*	ControlObjectType

Der komplexe Artefakttyp ObjectStructure ist in Tabelle 6-10 gezeigt.

Tabelle 6-10. Komplexer Artefakttyp ObjectStructure

Multiplizitaet	*aggregierter Artefakttyp*
0..*	ControlObjectType
0..*	Instantiation

Neben der Aggregation von ControlObjectType und Instantiation weist der Typ ObjectStructure zusätzlich eine topLevelControlObjectType-Relation zu ControlObjectType auf, mit welcher die Wurzel des Instanzenbaums markiert wird (in Tabelle 6-10 nicht gezeigt).

Als Nächstes müssen nun noch die abstrakten komplexen Artefakttypen für die Control-Object-Type-Dokumente beschrieben werden. Dafür wird zweistufig vorgegangen und in einer sog. ControlObjectTypeConfiguration (Tabelle 6-11) alle für ein Control-Object-Type jeweils nur lokal „sichtbaren“ Artefakttypen (z.B. Attribute) aggregiert.

Tabelle 6-11. Komplexer Artefakttyp ControlObjectTypeConfiguration

Multiplizitaet	*aggregierter Artefakttyp*
0..1	ControlObjectType
0..*	Attribute
0..*	Timer
0..*	Instantiation
0..*	Task
0..*	Strategy
0..*	Transition
0..*	State
0..*	AbstractStateSet
0..*	Condition

Tabelle 6-11. Komplexer Artefakttyp ControlObjectTypeConfiguration

Multiplizitaet	*aggregierter Artefakttyp*
0..*	Event
0..*	Action
0..*	Signal
0..*	ActualParameter

Eine solche lokale Sichtbarkeit bedeutet auch, dass dieselben Namen für jeweils andere Instanzen in anderen Objekttypen verwendet werden dürfen. Zum Beispiel können zwei unterschiedliche Objekttypen jeweils ein Attribut a definieren, das unterschiedliche Datentypen besitzt.

In einem umfassenden komplexen Artefakttyp System (Tabelle 6-12) werden dann alle solche ControlObjectTypeConfigurations zusammen mit den global „sichtbaren“ Artefakttypen (wie z.B. Signaltypen) aggregiert. Die globale Sichtbarkeit erfordert, dass hier alle Namen eindeutig sind. So darf es z.B. nur eine ControlObject-Type-Instanz mit dem Namen TempCtrl geben.

Tabelle 6-12. Komplexer Artefakttyp System

Multiplizitaet	*aggregierter Artefakttyp*
0..*	ControlObjectTypeConfiguration
0..*	DataType
0..*	SignalType
0..*	FormalParameter
0..*	Need

Zuletzt werden alle obigen komplexen Artefakttypen zu dem Artefakttyp RequirementsSpecification zusammengefasst (Tabelle 6-13).

Tabelle 6-13. Komplexer Artefakttyp RequirementsSpecification

Multiplizitaet	*aggregierter Artefakttyp*
0..1	Needs
0..1	TaskList
0..1	ObjectStructure
0..1	System

6.3 Ein Automatisierungsbeispiel

Bevor im zweiten Teil der Arbeit komplexe Aktivitäten automatisiert werden, wollen wir unsere bisher vorgestellten Techniken und Modelle an einem überschaubaren

Beispiel zur Anwendung bringen. Es soll dazu das *RoomAutomation*-System aus Abschnitt 6.1.3 auf Seite 138 automatisch instrumentiert werden, um die Testläufe der Prototypen auswerten zu können (in Abb. 6-8 auf Seite 146 hatten wir eine grafische Darstellung dieser Daten bereits gezeigt).

Dazu werden wir mir Hilfe unserer Aktionssprache AL++ (Abschnitt 4.2.2 auf Seite 65) und der in diesem Kapitel eingeführten Artefakte (Metatypen) des Produktmodells eine entsprechende Modelltransformation beschreiben. Die Transformation zwischen den abstrakten und konkreten Artefakten wollen wir dabei voraussetzen (siehe dazu Abschnitt 5.1.2 auf Seite 91) und stellen daher lediglich den Algorithmus zur Transformation der abstrakten Artefakte vor.

Die Instrumentierungsschritte sind dabei sehr einfach. Für jeden Sensor- oder Aktuator-Control-Object-Type müssen wir nur dafür sorgen, dass die empfangenen bzw. versendeten Signale protokolliert werden. Dazu werden wir die im Utils-Package (siehe Abschnitt 6.1.3 auf Seite 138) eingeführte Remote-Procedure printIt() verwenden, welche die Ausgabe eines Strings erlaubt und dabei stets die aktuelle Zeit (now) anhängt.

Die Transformationsalgorithmen wollen wir „top-down“ vorstellen und beginnen daher mit der Angabe des Aktions-Codes für den komplexen Artefakttyp RequirementsSpecification (siehe Tabelle 6-13 von oben).

```
1  /* RequirementsSpecification: */
2  public void instrument() {
3      ControlObjectType cot;
4      foreach(ControlObjectTypeConfiguration cotc in
5       this.itsSystem.itsControlObjectTypeConfiguration) {
6          cot = cotc.itsControlObjectType;
7          cot.instrument();
8      }
9  }
```

Da der komplexe Artefakttyp RequirementsSpecification die Control-Object-Types nicht direkt aggregiert, müssen wir zunächst über den komplexen Typ System und dann über die ControlObjectTypeConfigurations laufen.

Mit obiger Schleife rufen wir für jede Instanz eines ControlObjectTypes die folgende `instrument()`-Operation auf.

```
1  /* ControlObjectType: */
2  public void instrument() {
3      if(this.aggregatedInstantiation.size() > 0) {
4          return;
5      }
6      Strategy s;
```

```
7       foreach(Task t in this.implementedTask) {
8           s = t.realizingStrategy;
9           foreach(Transition tr in s.realizingTransition) {
10              tr.instrument();
11          }
12      }
13  }
```

Auch im ControlObjectType können wir noch keine Modellmodifikation vornehmen, da wir hier erst über die für unsere Modifikation interessanten Transitionen laufen müssen. Dazu besorgt man sich für jeden von einem Objekttyp implementierten Task dessen Strategie, von der aus man alle Transitionen durchlaufen kann. Zuvor wird allerdings geprüft (Zeile 3), ob es sich bei dem Objekttyp um einen Blattknoten und damit einen Sensor oder Aktuator handelt (Blattnoten besitzen keine aggregierten Instanzen). Sollte es sich bei einem Blattknoten nicht um einen Sensor oder Aktuator handeln, ist dies nicht kritisch, da in einem solchen Fall mehr Daten protokolliert würden als benötigt werden.

Die eigentliche Modifikation erfolgt letztlich in der `instrument()`-Operation des Transition-Typs:

```
1   /* Transition: */
2   public void instrument() {
3       foreach(Action act in effectedAction) {
4           act.actionNbr = act.actionNbr * 2;
5       }
6       Event ev = this.triggeringEvent;
7       if( (ev != null) && (ev.type == ReceiveEvent) ) {
8           ReceiveEvent re = (ReceiveEvent)ev;
9           Signal sig = re.creatingSignal;
10          RemoteCallAction rca = sig.computeAction();
11          if(rca != null) {
12              rca.actionNbr = 0;
13              this.effectedAction += rca;
14          }
15      }
16      foreach(Action ac in this.effectedAction) {
17          if(ac.type == SendAction) {
18              SendAction sa = (SendAction)ac;
19              Signal sig = sa.producedSignal;
20              RemoteCallAction rca = sig.computeAction();
21              if(rca != null) {
22                  rca.actionNbr = sa.actionNbr + 1;
23                  this.effectedAction += rca;
```

```
24                }
25            }
26        }
27    }
```

Die Instrumentierungsaktionen sollen jeweils direkt nach dem entsprechenden ReceiveEvent oder der SendAction eingefügt werden, um eine eventuelle Änderung von Attributbelegungen durch andere Aktionen auszuschließen. Daher werden die existierenden Aktionen in den Zeilen 3 bis 5 zunächst so um-nummeriert, dass zwischen jeder Aktion „Platz“ für mindestens eine weitere Aktion bleibt (wir nehmen an, dass actionNbr bei 1 beginnt).

Danach untersuchen wir die auslösenden Ereignisse und die ausgelösten Aktionen. Handelt es sich bei einem auslösenden Ereignis um einen ReceiveEvent (Zeile 7), also um einen Signalempfang, dann erzeugen wir eine neue Aktion zur Ausgabe unserer Instrumentierungsdaten und fügen diese zu der Menge der existierenden Aktionen hinzu (Zeile 13). Die Erzeugung der entsprechenden RemoteCallAction erfolgt mit Hilfe der `computeAction()`-Operation des Typs Signal (siehe unten).

Analog verfahren wir für die ausgelösten Aktionen. Ist unter diesen Aktionen eine SendAction, dann wollen wir die gesendete Information zu Instrumentierungszwecken ausgeben.

Als letztes ist nun noch die `computeAction()`-Operation von Signal genauer zu beschreiben:

```
1   /* Signal: */
2   public RemoteCallAction computeAction() {
3       SignalType sigType = this.itsSignalType;
4       Set fParams = sigType.itsFormalParameter;
5       Set aParams = this.itsActualParameter;
6       if( (fParams.size() == 1) && (aParams.size() == 1) ) {
7           FormalParameter fp;
8           foreach(fp in fParams) {}
9           DataType dt = fp.itsDataType;
10          ActualParameter ap;
11          foreach(ap in aParams) {}
12          String descr = "printIt(instName//"+
13           " '"+sigType.name+" '//"+
14           "(call "+dt.name+"ToStr("+ap.description+")),"+
15           " now)";
16          RemoteCallAction rca = new RemoteCallAction();
17          rca.description = descr;
18          return rca;
19      }
20      return null;
```

```
21  }
```

Zunächst bestimmen wir den SignalType des Signals, um dessen Liste an formalen Parametern zu erhalten. Zur Vereinfachung betrachten wir an dieser Stelle nur Signale mit exakt einem Parameter (was typisch für die Ein-/Ausgabesignale von Sensoren oder Akuatoren ist).

Da wir den Typ des Parameters für eine spätere Typkonversion zur Ausgabe benötigen, besorgen wir uns den formalen Parameter des Signaltyps. Hier nutzen wir einen kleinen „Trick“. Da wir wissen, dass die Menge `fParams` nur einen Eintrag beinhaltet können wir diesen mittels des `foreach`-Konstrukts sehr einfach auslesen (Zeile 8).

Genauso gehen wir für die aktuellen Parameter vor. Diese benötigen wir um zur Laufzeit den Wert des Signalparameters auszulesen. Damit haben wir alle notwendigen Daten zur Ausgabe mit der RPC printIt() zur Verfügung und müssen diese nur noch entsprechend zusammensetzen. Um später die Daten den jeweiligen Instanzen des Systems zuordnen zu können, geben wir als erstes den Instanzennamen aus, der in der vordefinierten Variable `instName` gespeichert ist.

Das Ergebnis der Ausführung der obigen Operationen auf eine Instanziierung des Produktmodells für unser *RoomAutomation*-Beispiel liefert für den Control-Object-Type RadiatorAct die in Abb. 6-23 gezeigte Transformation.

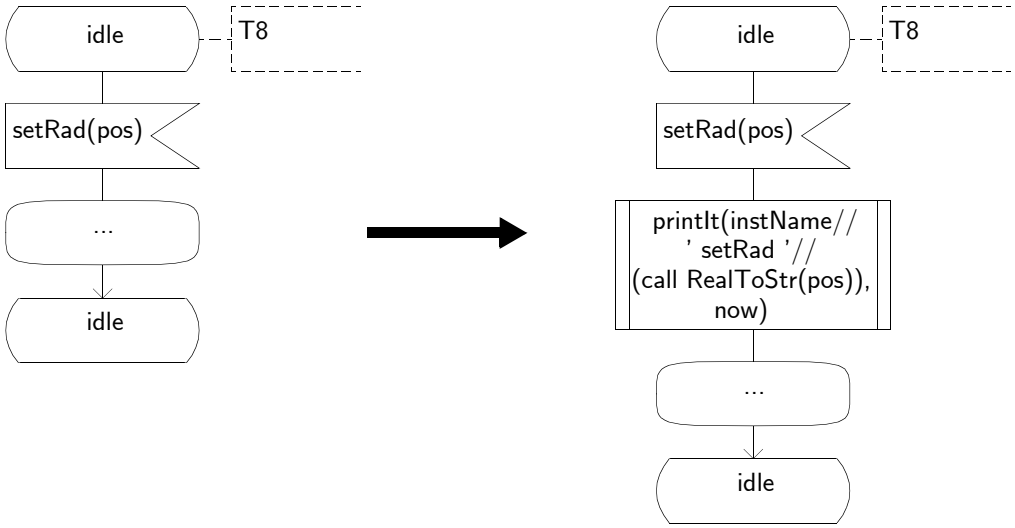


Abbildung 6-23. Instrumentierung am Beispiel von *RoomAutomation* (Objekttyp RadiatorAct)

Eine Ausführung diese SDL-Modells liefert schließlich die folgenden Daten:

```
...
ts1.tc1.obj1 newTemp 19.9983 at 1752.389804094
ra1.tc1.obj1 setRad 1.000000 at 1752.391470163
ra2.tc1.obj1 setRad 1.000000 at 1752.391678064
```

```
ts1.tc1.obj1 newTemp 20.0221 at 1762.399780958
ra1.tc1.obj1 setRad -1.000000 at 1762.401447981
ra2.tc1.obj1 setRad -1.000000 at 1762.401655167
...
```

Zusammenfassung

In diesem Kapitel wurde die Entwicklungsmethode PROBAnD zur Spezifikation reaktiver Systeme eingeführt. Neben der Einordnung in die Domäne der reaktiven Systeme und der Vorstellung der initialen PROBAnD-Methode wurde eine Erweiterung dieser Methode vorgestellt. Hauptmerkmal dieser Erweiterung ist die Spezifikation von erweiterten Endlichen Automaten durch die Komposition von Transitionen und die dadurch ermöglichte Verfolgbarkeit von Anforderungen bis hin zu diesen einzelnen Transitionen.

An einem kleinen Beispiel wurde schließlich die Anwendung der Methode und die Automatisierung einer einfachen Instrumentierungsaktivität vorgeführt. Interessant wird eine solche Automatisierung natürlich erst für große Beispiele oder komplexere Transformationen, auf welche wir im zweiten Teil der Arbeit eingehen werden.

— Teil II —

Qualitätssicherung durch Automatisierung

7 Qualitätssicherung durch Automatisierung

Qualität = Das Gegenteil des Zufalls.

— Klaus Zumwinkel (dt. Topmanager, Vorstandsvorsitzender der Deutschen Post AG, *1943)

Dieses Kapitel ist zunächst dem zentralen Begriff der Qualität gewidmet. Neben der Definition dieses Begriffs werden verschiedene Maßnahmen zur Qualitätssicherung diskutiert. Auf die konstruktiven Ansätze unter diesen Maßnahmen wird dann zum Abschluss des Kapitels eingegangen und gezeigt, wie diese im Rahmen der PROBAnD-Methode automatisiert wurden.

In den sich anschließenden Kapiteln 8 und 9 werden schließlich Ansätze zur Automatisierung weiterer Qualitätssicherungsmaßnahmen vorgestellt.

7.1 Qualität

Der Begriff „Qualität“ wird in DIN 55350 folgendermaßen definiert:

> **Definition 7-1:** „**Qualität** ist die Gesamtheit von Eigenschaften und Merkmalen eines Produkts oder einer Tätigkeit, die sich auf deren Eignung zur Erfüllung gegebener Erfordernisse bezieht.“ [DIN95]

Wie in [VIS04] richtig herausgestellt wird, handelt es sich hierbei um keinen universellen Begriff, da Qualität stets von der Perspektive abhängt, die durch eine Person und ihre individuellen Anforderungen in einer bestimmten Situation geprägt ist.

Die Qualität eines Software-Produkts kann insbesondere unter zwei Perspektiven gesehen werden, welche in den folgenden Definitionen (nach [VIS04]) präzisiert werden:

Definition 7-2: Die **externe Qualität** eines Software-Produkts ist Ausdruck dafür, in wie weit ein Produkt die expliziten und impliziten Anforderungen der *Benutzer* erfüllt. Die externe Qualität einer Software bezeichnet die Gesamtheit von Qualitäten, die sich auf die *Ausführung* der Software beziehen.

Definition 7-3: Die **interne Qualität** eines Software-Produkts ist Ausdruck dafür, in weit ein Produkt die Anforderungen der *Software-Entwickler* erfüllt. Die interne Qualität einer Software bezeichnet die Gesamtheit von Qualitäten der Software-Spezifikation, die sich *ohne* Berücksichtigung der konkreten Ausführungsbedingungen bestimmen lassen.

So wird ein Benutzer z. B. Wert auf eine einfache Bedienung des Software-Produkts legen, wohingegen der Entwickler an Aspekten wie z. B. der Wartbarkeit oder der Portierbarkeit interessiert ist.

Sowohl zwischen der internen und der externen Qualität als auch der Qualität der Erstellung des Produkts (Prozessqualität) lässt sich ein kausaler Zusammenhang wie er in Abb. 7-1 gezeigt ist feststellen.

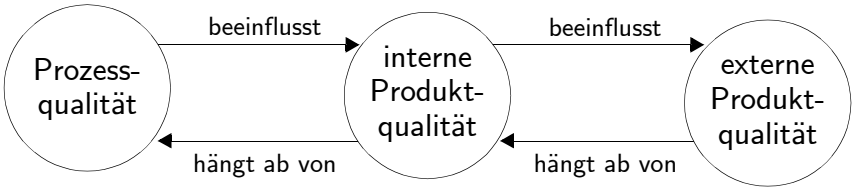


Abbildung 7-1. Zusammenhang zwischen Prozess- und Produktqualität (nach [ISO01])

In Vorwärtsrichtung beeinflusst die *Prozessqualität* (insbesondere der Reifegrad des Prozesses) die interne *Produktqualität*. Werden z. B. regelmäßig Qualitätssicherungsmaßnahmen – beispielsweise Inspektionen – durchgeführt, so erhöht sich dementsprechend die interne Qualität. Eine gute interne Qualität, die sich beispielsweise in der Einhaltung von Modellierungsrichtlinien oder in konsistenten Dokumenten zeigt, führt wiederum zu einer guten externen Qualität, weil eine Realisierung dann typischerweise mit weniger Fehlern erfolgt.

Bei obigen Zusammenhängen sollte man allerdings berücksichtigen, dass ein Produkt, das die internen Qualitätsanforderungen vollständig erfüllt, nicht automatisch auch seine externen Qualitätsanforderungen zu 100 % erreicht. Dies liegt an dem unterschiedlichen Fokus der Personen, die an internen und externen Qualitätsmerkmalen interessiert sind. Insbesondere bedeutet dies, dass man ohne eine Validierung

zusammen mit dem Kunden (siehe dazu Abschnitt 9.1.1 auf Seite 242) nicht sicher sein kann, die gewünschte externe Qualität zu erreichen.

Mit welchen Maßnahmen sich die einzelnen Qualitätseigenschaften nun gewährleisten lassen, wollen wir im Folgenden klären.

7.1.1 Maßnahmen zur Qualitätssicherung

Unter *Software-Qualitätssicherung* versteht man angemessene und aufeinander abgestimmte Maßnahmen zur Erfüllung vorgegebener Qualitätsanforderungen [RoB87, S.148]. Insbesondere sollten stets die beiden Arten von sich gegenseitig ergänzenden Maßnahmen umgesetzt werden [BrH93, S.333]:

1. *konstruktive Maßnahmen*, welche die fehlerfreie Erzeugung von Produkten zum Ziel haben („Correctness by Construction"), und
2. *korrektive* (*analytische*) *Maßnahmen*, die mit dem Ziel durchgeführt werden, Fehler aufzudecken und zu verbessern.

Werden Aktivitäten von Menschen ausgeführt, können Fehler nie ganz ausgeschlossen werden. Daher reichen die konstruktiven Maßnahmen alleine nicht aus, sondern müssen um korrektive Maßnahmen ergänzt werden.

Generell gilt, dass Qualitätssicherungsmaßnahmen so früh wie möglich in der Systementwicklung durchgeführt werden sollten, da spät gefundene Fehler einen enormen Korrekturaufwand nach sich ziehen. Pomberger et al. zitieren in [PoB93, S.46] z.B. ein Ergebnis von Boehm, das besagt, dass die Fehlerkorrektur während des Betriebs das fünfzigfache der Korrektur während der Systemspezifikation kostet. Auch Liggesmeyer verweist in [Lig02, S.29] auf ähnliche Zahlen, die in einer Untersuchung von Möller zu tage traten. Illgen nennt in [Ill04] aktuelle Zahlen aus dem Bereich der Automobilentwicklung. Dort kostet die Fehlerkorrektur einer Software-Komponente während der Serienproduktion das 86-fache der Korrektur während der Konzeptphase (Analysephase).

Allerdings gilt auch, dass jede Form der Qualitätssicherung trotz einer Automatisierung unter Umständen die Entwicklungskosten erhöhen kann. Wegen einer höheren Qualität der abgelieferten Produkte führt dies jedoch normalerweise zu einer deutlichen Reduktion der über den gesamten Lebenszyklus der Software anfallenden Kosten. So können nach Pomberger et al. [PoB93, S.175] insbesondere die Korrekturkosten, die 83% der Wartungskosten ausmachen, reduziert werden.

Im weiteren Verlauf dieser Arbeit werden wir Ansätze zur Realisierung beider Arten von Qualitätssicherungsmaßnahmen vorstellen. In diesem Kapitel werden wir dazu kurz auf mögliche konstruktive Maßnahmen eingehen, welche durch eine Automatisierung im Rahmen der PROBAnD-Methode möglich werden. In Kapitel 8 werden korrektive Maßnahmen auf der Basis statischer Analysen vorgestellt, welche

der Sicherstellung der internen Produktqualität dienen. Im anschließenden Kapitel 9 werden wir verschiedene Prototyping-Techniken einführen, welche sowohl der Sicherstellung der internen Produktqualität (durch dynamische Analysen) als auch der Sicherstellung der externen Qualität (durch Validierung) dienen, wobei die automatische Prototyp-Erstellung als konstruktive Maßnahme vorgestellt wird.

Zunächst wollen wir aber einen genaueren Überblick über beide Arten von Maßnahmen geben.

Maßnahmen zur Sicherung der internen Qualität

Merkmale, die es bei der Gewährleistung der internen Qualität zu berücksichtigen gilt, sind insbesondere (zum Teil nach [Tra93, S. 25ff.], [VIS04], [Lig02, S. 7ff.]):

- *Korrektheit*: Ein Produkt ist korrekt, wenn es seiner Spezifikation entspricht.
- *Vollständigkeit* und *Konsistenz*: Voraussetzung für die Korrektheit ist, dass ein Artefakt vollständig und konsistent ist. Auf diese Merkmale werden wir detailliert in Abschnitt 8.1 auf Seite 188 eingehen.
- *Verständlichkeit*: Darunter versteht man die Eignung der Artefakte mit möglichst geringem Aufwand verstanden zu werden.
- *Wartbarkeit*: Ein Software-Produkt ist wartbar, wenn Anpassungen und Änderungen auf Grund neuer Anforderungen oder veränderter Umgebungsbedingungen einfach durchzuführen sind.
- *Größe*: Die Größe eines Artefakts beschreibt dessen („physikalische“) Ausdehnung (z. B. in Anzahl Zeilen eines Programms).
- *interne Komplexität*: Neben der internen Komplexität, welche eine psychologische Komplexität eines Artefakts beschreibt, existiert auch eine externe Komplexität (s. u.).

Viele der obigen Merkmale können automatisch bestimmt und die Sicherung dieser Qualitätseigenschaften unterstützt werden. So werden wir in dieser Arbeit sowohl auf die automatische Bestimmung der Korrektheit, der Konsistenz und der Vollständigkeit (in Kapitel 8 und zum Teil in Kapitel 9) als auch auf die automatische Messung der Größe und der internen Komplexität (in Kapitel 10) eingehen.

Maße wie z. B. „Verständlichkeit“ aber auch „Wartbarkeit“ sind dabei schwierig automatisch zu bestimmen, da sie oft von dem subjektiven Eindruck des Entwicklers abhängen. Auch können viele der Korrektheitsprüfungen nur manuell durchgeführt werden, da diese ein Verständnis der Semantik der Modellelemente voraussetzen. So ist es z. B. nicht möglich, für jedes M_1-Modell der PROBAnD-Methode automatisch zu prüfen, ob damit eine Gebäudesteuerung sinnvoll spezifiziert wurde. Unsinnig wäre z. B. eine Helligkeitsregelung in einem Raum ohne Fenster und Türen (die Begriffe „Fenster“, „Tür“ und „Raum“ wurden auf M_1 definiert weshalb deren

„Verwendung“ nicht durch Constraints auf Ebene M_2 ausgedrückt werden kann, vgl. dazu auch Abschnitt 8.1 auf Seite 188).

Eine Sicherung solcher Qualitätsmerkmale muss daher insbesondere durch (manuelle) Inspektionen (Reviews) mit ihren unterschiedlichen Ausprägungen erfolgen. Liggesmeyer stellt diese Ausprägungen in [Lig02, S. 285ff.] vor und weist insbesondere darauf hin, dass Inspektionen andere Prüftechniken (insbesondere automatische Analysen) ergänzen. Die in dieser Arbeit vorgestellten Maßnahmen zur Qualitätssicherung sind alleine also nicht ausreichend. Es sollte daher immer ein angemessener Mix von Qualitätssicherungsmaßnahmen zur Anwendung gebracht werden.

Maßnahmen zur Sicherung der externen Qualität

Wie wir weiter oben schon erläuterten, kann die Sicherstellung der externen Qualität immer nur zusammen mit dem Kunden erfolgen, da er diejenige Person ist, deren Qualitätsanforderungen es zu erfüllen gilt.

Wichtige Qualitätsmerkmale der externen Qualität sind daher (nach [May90, S. 89], [VIS04] und [Lig02, S. 7ff.]):

- *Zuverlässigkeit*: Wir wollen den Zuverlässigkeitsbegriff in dieser Arbeit im weiteren Sinne als die Fähigkeit eines Systems verstehen, den Vorstellungen (Anforderungen) des Benutzers zu genügen (siehe auch [Que02, S. 169] und [Jal97, S. 11]). Neben diesem allgemeinen Begriff der Zuverlässigkeit (engl. „dependability“) existiert auch eine Verwendung von „Zuverlässigkeit“ im engeren Sinne (engl. „reliability“). Diese eingeschränktere Zuverlässigkeit gibt die Wahrscheinlichkeit an, dass ein Produkt während einer bestimmten Zeitspanne kein Fehlverhalten aufweist (vgl. [Lig02, S. 9] und [Jal97, S. 545]).
- *Verlässlichkeit*: Die Verlässlichkeit definieren wir als den Grad, zu welchem das System die gewünschten Funktionen im Falle eines Hardware- oder Softwaredefekts zur Verfügung stellen kann.
- *Effizienz*: Die Effizienz gibt an, wie ressourcenschonend die Funktionalität des Systems realisiert wurde. Sie hängt eng mit der externen Komplexität zusammen.
- *externe Komplexität*: Anders als die interne Komplexität beschreibt die externe Komplexität die Zeit- bzw. Speicherkomplexität einer Software (eines Algorithmus), also wie aufwändig die Berechnung in Abhängigkeit der Problemgröße ist.
- *Robustheit*: Die Robustheit ist eine Eigenschaft des Systems auch in ungewöhnlichen (oder nicht-spezifizierten) Situationen definiert zu arbeiten und sinnvolle Reaktionen durchzuführen.

- *Verstehbarkeit*: Darunter fasst man die Eigenschaft auf, dass ein Softwaresystem oder dessen Teile (z.B. eine Benutzeroberfläche) mit möglichst geringem Aufwand verstanden werden können. Sie hängt eng mit der Benutzbarkeit zusammen.

Die wichtigste korrektive Maßnahme zur Sicherstellung dieser Qualitätseigenschaften (insbesondere von Zuverlässigkeit und Verstehbarkeit) ist dabei in unseren Augen das Prototyping, das wir – wie bereits erwähnt – in Kapitel 9 einführen werden.

Daneben können aber auch hier wieder Inspektionen zusammen mit dem Kunden durchgeführt werden, was insbesondere sehr früh im Prozess angebracht ist, wenn noch keine Prototypen oder Systemteile existieren.

7.2 Automatisierung konstruktiver Maßnahmen

Konstruktive Maßnahmen sind eine wichtige Säule einer umfassenden Qualitätssicherung. Wir werden daher in diesem Abschnitt auf einen Teil der von uns gewählten Realisierung solcher Maßnahmen eingehen. In den folgenden Kapiteln werden vereinzelt weitere, speziellere, solcher Qualitätssicherungsansätze vorgestellt.

Bei der Vorstellung der PROBAnD-Methode wurden in Abschnitt 6.1.2 auf Seite 135 die typischen Schritte zur Überführung einer Problembeschreibung in eine Anforderungsspezifikation vorgestellt. Einen großen Anteil an diesen Aktivitäten nimmt die Übertragung von Entwicklungsinformationen aus einem Dokument in eine anderes Dokument ein. So müssen z.B. die für einen Task in der Taskliste beschriebenen Eigenschaften name, description und die realisierten Anforderungen (realizedBy-Relation) in das Dokument des Control-Object-Types übertragen werden, welches diesen Task implementiert (vgl. dazu auch das Beispiel aus Abschnitt 6.1.3 auf Seite 138).

Durch eine Automatisierung dieser „stupiden“ Tätigkeiten kann man einen Effizienz- und Qualitätsgewinn erreichen, da die Überführung vom Werkzeug fehlerfrei und mit sehr geringem Aufwand ausgeführt werden kann (siehe dazu die Resultate in Abschnitt 10.2.2 auf Seite 316).

Die Realisierung einer solchen automatischen Überführung ist mit der Verwendung der in Abschnitt 5.1.2 auf Seite 91 eingeführten Parse- und Unparse-Vorgänge sehr einfach darzustellen. Zunächst werden die relevanten Eingangsdokumente geparsed und man erhält einen Satz abstrakter Artefakte. Aus diesen kann man nun durch einen Unparse-Schritt die gewünschten Zieldokumente erhalten. Für obiges Beispiel würde man also zunächst die Taskliste parsen und danach die Dokumente der einzelnen Control-Object-Types (die durch die implements-Relation in der Taskliste beschrieben werden) erzeugen. Man führt also eine Kette von Modelltransformationen der folgenden Form durch:

$$M_{\mathrm{e}} \xrightarrow{\mathrm{0F}} M_{\mathrm{i}} \xrightarrow{\mathrm{0F}} M_{\mathrm{a}}$$

M_{e} sind dabei die Dokumente, die das Eingabe-, und M_{a} sind die Dokumente, die das Ausgangsmodell beschreiben. Bei M_{i} handelt es sich um die abstrakten Artefakte, die in einer internen Datenstruktur der Werkzeuge abgelegt werden.

Neben dem „Voranbringen“ der Entwicklung durch die Generierung von Dokumenten einer späteren Phase, kann man mit einem solchen Ansatz auch die Dokumente einer früheren Phase aus den aktuell bearbeiteten Dokumenten erzeugen. Damit wird es möglich, stets ein konsistentes Bild der Entwicklung zu halten (insbesondere mit der in Abschnitt 8.1 auf Seite 188 vorgestellten automatischen Konsistenzprüfung), womit die „frühen“ Dokumente auch nach Abschluss der Entwicklung als sinnvolle Dokumentation des endgültigen Systems verwendbar sind und damit die Wartung stark vereinfacht wird.

In meinen Augen ist ein solches Vorgehen, also die *automatische* Konsistenthaltung der Dokumente, einem „agilen“ Vorgehen vorzuziehen. Bei den agilen Methoden (siehe z.B. [Coc02]) wird auf die Erstellung von allem, was die eigentliche Software-Entwicklung nicht voranbringt, verzichtet. Damit werden letztlich aber nur die Wartungskosten zu Gunsten der aktuellen Entwicklungsproduktivität geopfert [Mau02].

7.2.1 Konstruktive Maßnahmen im Kontext der PROBAnD-Methode

Die Artefakte der PROBAnD-Methode sind alle Teil ein und derselben Begriffswelt (siehe [Hol03, S.6ff.]), die durch das PROBAnD-Produktmodell ausgedrückt wird. Die Dokumente und Views sind daher derart gestaltet, dass sie jeweils eine direkte Repräsentation der abstrakten Artefakte darstellen. Eine M0-Transformation des M_{i} Modells ist folglich – bis auf eine Ausnahme, die wir weiter unten erläutern werden – nicht notwendig und vereinfacht die konstruktiven Aktivitäten daher sehr.

Im Allgemeinen ist eine solche vertikale Transformation, die das Modell näher zur Implementierungsebene bringt, allerdings nicht unproblematisch, da sich der Eingangs- und der Ausgangsformalismus bzgl. der beschriebenen Konzepte unterscheiden können. Wir werden dieses Problem in Abschnitt 9.1.2 auf Seite 246 für das Prototyping diskutieren.

Die Ansätze des Parsens und Unparsens, welche für die hier vorgestellte konstruktive Maßnahme zentral sind, wurden bereits in Abschnitt 5.1.2 auf Seite 91 eingeführt und erläutert. Wir wollen dies hier nicht weiter vertiefen, da die eingesetzten Realisierungstechniken (wie z.B. die Erzeugung der Syntax-Parser mit Parser-Generatoren) Stand der Technik sind.

Wichtig für eine korrekte Anwendung der Dokumentgenerierung im Rahmen der PROBAnD-Methode ist allerdings, die jeweils in den Dokumenten beinhaltete Entwicklungsinformation und deren Anteil an den gesamten Modellkonzepten zu kennen. Ansonsten könnten bei einer falschen Anwendung der Werkzeuge wichtige Modellinformationen verloren gehen.

In Abb. 7-2 ist zunächst eine Übersicht über die in den PROBAnD-Dokumenten enthaltenen konkreten Artefakttypen und deren Überschneidungen gezeigt.

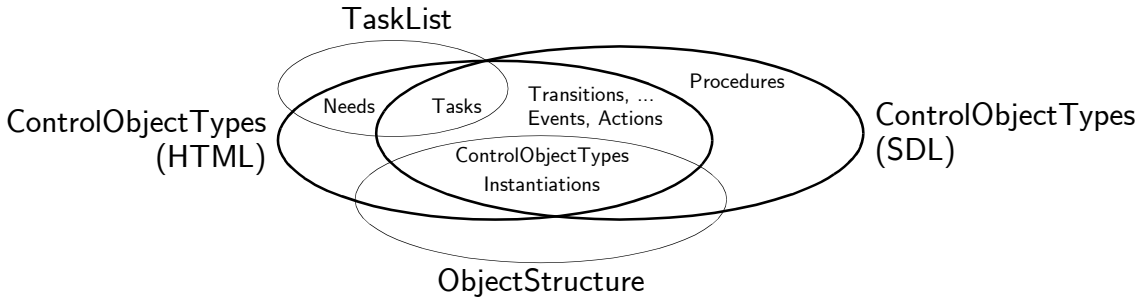


Abbildung 7-2. Repräsentation der Modellinformation in den konkreten Artefakten

Wie man erkennt, werden die Needs nur in den HTML-Control-Object-Type-Dokumenten und der Taskliste beschrieben, wohingegen die Procedures (lokal und remote) nur in SDL dargestellt werden.

Will man also aus den SDL-Dokumenten (die einen späteren Entwicklungsstand darstellen) die „früheren“ HTML-Dokumente erzeugen, so müssen die Need-Artefakte bekannt sein. Dies lässt sich nur bewerkstelligen, wenn neben den SDL-Dokumenten auch die aktuellen HTML-Dokumente (zumindest die Taskliste) geparsed werden.

Umgekehrt sind in HTML zunächst keine Procedures beschrieben (in den Control-Object-Type-Dokumenten aus Abschnitt 6.1.3 auf Seite 138 existiert keine entsprechende Eintragung). Dies impliziert, dass sobald man eine Prozedur in SDL definiert hat, bei jedem konstruktiven Schritt immer auch die SDL-Spezifikation geparsed werden muss. Da die Handhabung solcher Abhängigkeiten nicht unbedingt praktikabel ist, entschlossen wir uns an dieser Stelle eine Ergänzung der HTML-Dokumente vorzunehmen, s.d. der Inhalt der Prozeduren zwar abgelegt aber nicht verändert werden kann. Damit müssen immer nur bei einer Änderung in SDL die SDL-Dokumente geparsed werden, ansonsten kann man die HTML-Dokumente losgelöst von SDL bearbeiten.

Erzeugung von Signallisten in SDL

Wie wir bereits oben erwähnten, können fast alle konkreten Artefakte direkt aus den abstrakten Artefakten abgeleitet werden. Einzige Ausnahme bildet die Erzeugung der Signallisten in den SDL-Control-Object-Types. Diese verlaufen zwischen

den einzelnen Instanzen und müssen die Namen der über diese Kanäle versendeten Signaltypen beinhalten (siehe dazu das Beispiel aus Abb. 6-6 auf Seite 145).

Die Information, welche Signalnamen in den jeweiligen Signallisten auftauchen müssen, ist nicht explizit in den abstrakten Artefakten beschrieben. Sie lässt sich aber durch ein Verfolgen der produces- und consumes-Relationen nach folgender einfachen Regel ableiten: Immer dann wenn ein Control-Object-Type ein Signal von einem aggregierenden Objekttyp empfangen kann, muss dieses Signal in die Inp-Signalliste eingetragen werden (siehe Abb. 6-6). Ein Objekttyp kann ein Signal empfangen, wenn mindestens ein Task existiert, welcher den Signaltyp konsumiert (consumes-Link) und mindestens ein Task im aggregierenden Objekttyp existiert, welcher diesen Signaltyp produziert (produces-Link). Analog wird die Outp-Signalliste bestimmt.

In einer AL++-Realisierung stellt sich dies wie folgt dar:

```
1   /* ControlObjectType: */
2   protected Set inp = new HashSet();
3   public Set getInp() {
4       return inp;
5   }
6
7   public void determineInpSignalLists() {
8       foreach(Task t in implementedTask) {
9           Strategy s = t.realizingStrategy;
10          foreach(SignalType st in s.producedSignalType) {
11              foreach(Instantiation inst
12               in this.aggregatedInstantiation) {
13                  ControlObjectType aCot =
14                   inst.instantiatedControlObjectType;
15                  foreach(Task at in aCot.implementedTask) {
16                      Strategy as = at.realizingStrategy;
17                      foreach(SignalType ast
18                       in s.consumedSignalType) {
19                          if(st == ast) {
20                              aCot.inp.add(st);
21                          }
22                      }
23                  }
24              }
25          }
26      }
27      foreach(Instantiation inst in this.aggregatedInstantiation) {
28          ControlObjectType aCot =
```

```
29              inst.instantiatedControlObjectType;
30             aCot.determineInpSignalLists();
31         }
32   }
```

Für jede ControlObjectType-Instanz wird eine Menge `inp` definiert, welche die Signaltypen der Inp-Signalliste speichert.

Wird die Operation `determineInpSignallists()` von der obersten ControlObjectType-Instanz (topLevelControlObjectType im komplexen Artefakttyp ObjectStructure) aufgerufen, dann wird der gesamte Instanziierungsbaum rekursiv durchlaufen (Zeilen 27–31) und die gewünschte Information berechnet (Zeilen 8–25).

Für jeden besuchten Control-Object-Type wird dazu zunächst bestimmt, welche Signaltypen produziert werden (indem alle Tasks und Strategien, welche vom jeweiligen Objekttyp implementiert werden, durchlaufen werden). Für jeden einzelnen dieser Signaltypen wird dann überprüft, ob er in einem der aggregierten Objekttypen konsumiert wird (also ob ein Task existiert, welcher diesen Signaltyp konsumiert). Ist dies der Fall, dann wird der Signaltyp in die Eingangssignalliste des jeweiligen aggregierten Objekttyps `aCot` eingetragen.

Nachdem der gesamte Baum durchlaufen wurde, kann dann durch den Zugriff auf die `inst`-Menge (mit `getInp()`) für jeden Objekttyp die gewünschte Signalliste abgefragt und die entsprechenden SDL-Views erzeugt werden.

Grenzen des Ansatzes

Trotz der oben so optimistisch geschilderten Anwendung dieser Technik, stößt man auch bei einem dokumentzentrierten Ansatz an Grenzen. Bei der Einführung automatischer Konsistenzprüfungen in Abschnitt 8.1 auf Seite 188 werden wir einige dieser Probleme diskutieren.

Relevant an dieser Stelle ist das Problem, dass man nicht zuverlässig entscheiden kann, welche Änderungen der Dokumente M_{e} in das konsistente Gesamtmodell M_{i} einfließen sollen, aus welchem dann die neuen Dokumente M_{a} generiert werden. Sind z.B. in den HTML- und in den SDL-Dokumenten widersprüchliche Views spezifiziert, z.B. eine unterschiedliche Anzahl von Parametern für denselben Signaltyp, so kann man zunächst nicht entscheiden, welcher der Views die korrekte Information beinhaltet. Dies ist ein allgemeines Problem im jedem dokumentzentrierten Ansatz, welcher Inkonsistenzen zulässt.

Die Entscheidung kann nicht zuverlässig durchgeführt werden, weil es zum einen möglich ist, dass noch gar kein korrektes Artefakt existiert, da sich die Entwicklung in einem inkonsistenten Zustand befindet (z.B. während einer länger andauernden Änderung, siehe Abschnitt 5.1 auf Seite 82). Zum anderen, ist es nicht unbedingt gesagt, dass die Dokumente der späteren Phase (hier also SDL) die korrekten Arte-

fakte beinhalten. Man könnte ebenso eine Änderung in HTML durchgeführt haben, um dadurch ein fehlerhaftes Artefakt zu korrigieren.

Man könnte nun zur Vereinfachung den Fall, der typischerweise während der Entwicklung auftritt, einer solchen (automatisierten) Entscheidungsstrategie zu Grunde legen. Die Festlegung dieses Falls fällt in meinen Augen allerdings sehr schwer. Da das Werkzeug nicht entscheiden kann, ob eine Abweichung von diesem typischen Fall vorliegt, müsste dies der Entwickler zusätzlich bei jedem Aufruf der Werkzeuge angeben. Wir gehen daher einen anderen Weg, bei welchem wir den Modellierer über alle Unterschiede zwischen den Views in Kenntnis setzen und es ihm überlassen zu entscheiden, ob diese kritisch sind oder nicht und welche Artefakte eventuell angepasst werden müssen.

Zusammenfassung

Der Begriff der „Qualität“ wurde in diesem Kapitel eingeführt und verschiedene Qualitätseigenschaften eines Softwareprodukts definiert. Desweiteren wurde eine Unterscheidung zwischen korrektiven und konstruktiven Qualitätssicherungsmaßnahmen vorgenommmen und die wichtigste konstruktive Maßnahme, die im Rahmen der PROBAnD-Methode eingesetzt wurde, erläutert.

8 Automatische Analyse und Messung von Software-Modellen

Einen Fehler machen und ihn nicht korrigieren – das erst heißt wirklich einen Fehler machen.

— Konfuzius (551–479 v. Chr.)

Eine Möglichkeit, die interne Qualität von Software sicherzustellen, ist der Einsatz *statischer Analysen*. Statische Analysen können ohne eine Ausführung der Software (wie sie z. B. beim Prototyping erfolgt, vgl. Kapitel 9) durchgeführt werden [Lig02, S. 40] und somit bereits sehr früh in der Entwicklung zur Qualitätssicherung beitragen. In diesem Kapitel werden wir unterschiedliche Realisierungen solcher Analysen aufzeigen und deren Automatisierung erläutern.

Die in einem dokumentzentrierten Ansatz wichtigste Form der statischen Analyse ist die Identifikation unvollständiger oder inkonsistenter Dokumente. Daher stellen wir im weiteren Verlauf dieses Kapitels zunächst einen Ansatz für die automatische Detektion und – wenn möglich – Korrektur solcher Inkonsistenzen vor, wobei zur Automatisierung an vielen Stellen generische Lösungen auf der Basis der verbesserten Multiebenenmodellierung und der AL++ (vgl. Kapitel 4) vorgestellt werden.

Sind solche Aktivitäten noch recht einfach und zum Teil auch manuell durch einfache Inspektionstechniken zu unterstützen, ist die Durchführung für anspruchsvollerer Prüfungen ohne eine Automatisierung kaum denkbar. Als Vertreter solcher komplexer Analysen werden wir die Detektion von Interaktionen zwischen Features (Produktmerkmalen oder Anforderungen) vorstellen. Solche Feature-Interaktionen können zu einem unerwünschten Systemverhalten führen und dadurch die Qualität negativ beeinflussen.

Zum Abschluss dieses Kapitels werden die automatische Bewertung von Artefakten (durch eine Messung von Artefakteigenschaften) und die Visualisierung von Modellinformationen vorgestellt.

8.1 Behandlung inkonsistenter und unvollständiger Dokumente

Bereits in Abschnitt 5.1 auf Seite 82 im ersten Teil dieser Arbeit hatten wir als wichtige Eigenschaft eines dokumentzentrierten Ansatzes die möglichen Inkonsistenzen zwischen Dokumenten herausgestellt. Auch wenn solche Inkonsistenzen während der Entwicklung an vielen Stellen wichtig sind und auch produktiv sein können, sollte mit dem Ziel einer hohen Produktqualität darauf hingearbeitet werden, nach gewissen Entwicklungsschritten eine konsistente Gesamtspezifikation und am Ende der Entwicklung eine vollständige Spezifikation zu erhalten.

In diesem Abschnitt werden wir zeigen, wie mit Hilfe unseres Automatisierungsansatzes eine solche Prüfung auf Konsistenz und Vollständigkeit realisiert werden kann und welche Möglichkeiten der automatischen Korrektur von Inkonsistenzen oder Unvollständigkeiten zur Wahl stehen. Das Ziel dieses Teils der Arbeit war allerdings nicht, neue oder alternative Lösungen für Konsistenzprobleme zu finden, sondern die Notwendigkeit, die Modellkonsistenz zu garantieren, um eine durchgängige Werkzeugkette realisieren zu können. Deshalb halten wir einen Vergleich unserer Ansätze mit bereits existierenden Verfahren an dieser Stelle für nicht zwingend.

Bevor wir auf unsere Lösungen eingehen, sollen die bereits verwendeten Begriffe „Konsistenz“ und „Vollständigkeit“ zunächst definiert werden (vgl. [Hei04], [VIS04], [LaC04] und [HHS02]):

Definition 8-1: Eine Spezifikation ist **vollständig**, falls keine Angaben in der Spezifikation fehlen.

Definition 8-2: Eine Spezifikation ist **konsistent**, falls keine mehrdeutigen oder widersprüchliche Stellen in der Spezifikation existieren.

In der Praxis fällt eine eindeutige Zuordnung eines „Fehlers“ zu einem der beiden Begriffe nicht immer leicht. So kann man sich offensichtlicherweise bei einer fehlenden Information (Unvollständigkeit) auch eine mehrdeutige Interpretation dieser fehlenden Stelle vorstellen (Inkonsistenz). Auch Lange et al. identifizieren in [LCM03, S. 26] ein solches Problem. Für den weiteren Verlauf werden wir daher den Begriff „unvollständig“ unter dem Begriff „inkonsistent“ subsumieren.

8.1.1 Klassifikation von Inkonsistenzen

Um zu verstehen, welche Arten von Inkonsistenzen in einer Spezifikation auftreten können und wie diese durch Werkzeuge detektiert (und evtl. korrigiert) werden können, beginnen wir mit einer Klassifikation solcher Inkonsistenzen. Abb. 8-1 zeigt dazu eine Taxonomie, die sich für unsere Arbeit bewährt hat.

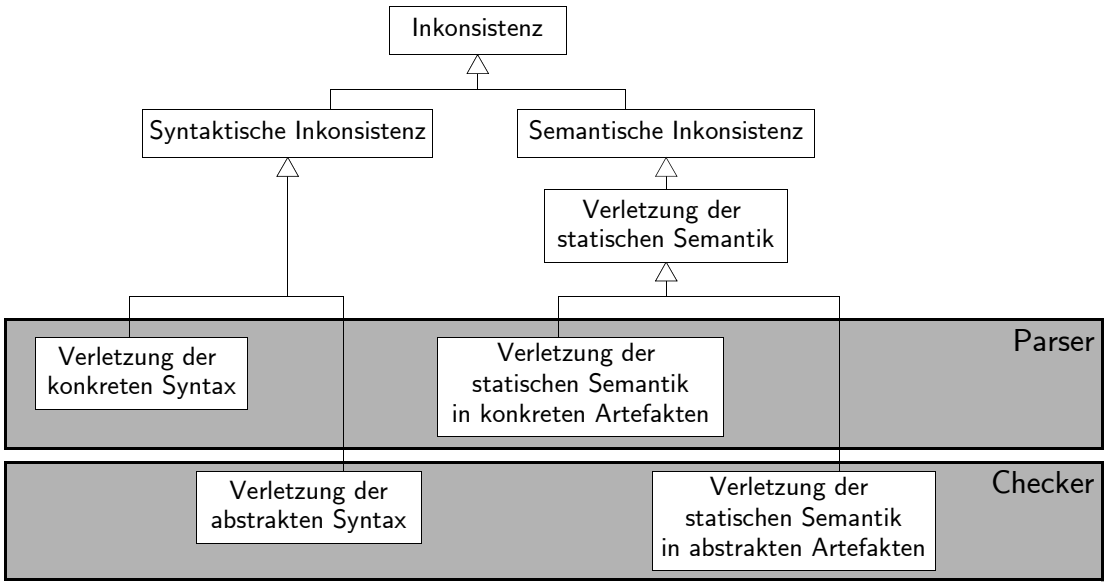


Abbildung 8-1. Klassen von Inkonsistenzen und Werkzeuge zur Detektion

Neben der Einordnung der Inkonsistenzen wird in der Abbildung auch gezeigt, welches unserer Werkzeuge für die Behandlung der Inkonsistenzen zuständig ist. Da nach dem Parse-Vorgang nur noch auf abstrakten Artefakten gearbeitet wird, muss die Konsistenzprüfung der konkreten Artefakte bereits im Parser erfolgen. Die Überprüfung der abstrakten Artefakte wurde in ein separates Werkzeug, den *Checker* ausgelagert. Da abstrakte Artefakte nicht nur der internen Darstellung des Parsers dienen sondern auch die Basis unserer M0-Transformationen bilden (siehe Abschnitt 5.1.2 auf Seite 91), kann so nach jeder Modelltransformation durch einen Aufruf des Checkers die Konsistenz der erzeugten Spezifikation geprüft und in gewissen Grenzen automatisch hergestellt werden.

Verletzung der konkreten Syntax

Bei der ersten Klasse von Syntaxverletzungen handelt es sich um konkrete Artefakte (oder Dokumente), die nicht der geforderten konkreten Syntax entsprechen. Ein Beispiel aus dem Kontext der PROBAnD-Methode wäre die Verwendung des Zeichens „*“ zur Trennung von Zuständen in einer HTML-Transition an Stelle des geforderten Doppelpunktes (zur Grammatik vgl. Abschnitt 6.2.3 auf Seite 151).

Demzufolge wird ein solcher Fehler beim Aufbau des abstrakten Syntaxbaums detektiert, was in der „Syntax-Parsen"-Aktivität (siehe Abb. 5-8 auf Seite 93) des Parsers erfolgt. Je nach Schwere des Fehlers kann der Parse-Vorgang fortgesetzt oder muss er abgebrochen werden. Ein Abbruch wird z.B. bei obigem Fehler (Verwechslung der Trennzeichen) notwendig. Läge beim Parsen der HTML-Dokumente von PROBAnD z.B. nur eine falsche Schachtelung der HTML-Tags vor (Beispiel: `<tr><td></tr></td>`), so wäre eine automatische Korrektur dieser Syntax-Inkonsistenzen in Grenzen möglich. Man könnte hierfür z.B. die Konzepte der Internet-Browser, die in dieser Hinsicht robust realisiert sind, aufnehmen. Wegen der großen Gefahr der Fehlinterpretation bei einer solchen automatischen Korrektur wurde dies in unsere Arbeit allerdings nicht getan.

Verletzung der abstrakten Syntax

Ein typisches Beispiel für die Verletzung der abstrakten Syntax ist, dass man einen Link erzeugt, der eine Instanz des falschen Typs am Link-Ende besitzt. So wäre eine PROBAnD-Produktmodellinstanz, bei der eine Need-Instanz über die implements-Relation mit einer ControlObjectType-Instanz verbunden ist, nicht konsistent. Unser Ansatz lässt die Erzeugung solcher inkonsistenter Modelle gar nicht erst zu, da bei der Erzeugung von Links und Attributbelegungen stets die Typ-Konformität geprüft wird. Würde man also versuchen das obige Beispiel in einem Automatisierungswerkzeug zu implementieren, würde spätestens zur Laufzeit des Werkzeugs eine entsprechende Exception (Ausnahme) ausgelöst.

Ein Grenzfall der Nichteinhaltung der statischen Syntax stellt die Verletzung der für die Relationen angegebenen Multiplizitäten dar. Auch hier verbietet unser Ansatz, dass z.B. bei einer zu-1-Relation mehr als eine Instanz am Link-Ende referenziert werden kann. Andere Formen von Constraints bzgl. der Link-Enden können aber nicht forciert werden, so z.B. dass höchstens n oder mindestens m Instanzen am Ende vorhanden sein müssen. In unseren Augen handelt es sich hierbei bereits um Verletzungen der statischen Semantik.

Verletzung der statischen Semantik in abstrakten Artefakten

In Kapitel 2 (Abb. 2-1 auf Seite 10) wurde auf die Rolle der statischen Semantik (oder der „Well-formedness"-Regeln) als Einschränkung (Constraint, vgl. Abschnitt 3.1.3 auf Seite 37) der Syntaxelemente eingegangen, die sich auf Grund der semantischen Interpretation der Modellelemente ergibt.

Die obigen Multiplizitätsgrenzen würde man hier einordnen, da sich diese Grenzen aus der Bedeutung der Elemente ergeben. So macht in der PROBAnD-Methode ein Task eben nur dann Sinn, wenn er einem Objekttyp zugeordnet werden kann,

also die zu-1-Relation von Task zu ContolObjectType (implements) belegt ist und damit die untere Grenze der Multiplizität von „1“ eingehalten wird.

Weitergehende Einschränkungen sind z.B., dass man in einem „sinnvollen“ Modell fordern muss, dass jedes Attribut, das von einer Strategie gelesen wird, auch von mindestens einer Strategie geschrieben wird. Da Attribute allerdings auch als Konstanten eingesetzt werden können (wenn das Attribut defaultValue im Produktmodell belegt ist), sollte dieses Constraint weiter präzisiert werden. Als logischer Ausdruck könnte sich dies wie folgt darstellen (ein unbelegtes defaultValue-Attribut ist durch einen Leer-String gekennzeichnet):

$$\forall a@\mathsf{Attribute} : |\mathsf{writingStrategy}(a)| = 0 \rightarrow \mathsf{defaultValue}(a) \neq \texttt{""}$$

Eine Detektion solcher Arten von Inkonsistenzen erfolgt dann durch eine Auswertung der logischen Ausdrücke für die jeweils betrachtete Instanz des Modells.

Ob eine automatische Korrektur realisierbar ist, hängt von der Art der Verletzung ab. Wird z.B. ein atomares Artefakt nicht von dem relevanten komplexen Artefakt aggregiert (als Beispiel sei hier ein fehlender Task in der TaskList genannt), dann lässt sich dies sehr leicht automatisch korrigieren. Von dieser letzten Form der Korrektur wird in unserem Ansatz intensiv Gebrauch gemacht.

Verletzung der statischen Semantik in konkreten Artefakten

Zuletzt bleibt noch die Verletzung der „Well-formedness“-Regeln in konkreten Artefakten zu betrachten.

Als Beispiel einer ersten Art von Problemen kann hier die mehrfache Verwendung desselben Namens für unterschiedliche Tasks angeführt werden. In der konkreten Syntax ist eine solche Verwendung zunächst nicht ausgeschlossen, da zur Benennung eines Tasks lediglich ein Identifier (also Text) gefordert wird. Daher muss die konkrete Syntax weiter eingeschränkt werden, weshalb eine solche Klasse von Inkonsistenzen als Verletzung der konkreten „Well-formedness“-Regeln eingeordnet wird. Folglicherweise müssen solche Inkonsistenzen – wie auch die Verletzung der konkreten Syntax – bereits während des Parsens detektiert und behandelt werden.

Eine weitere Form solcher Verletzungen kann auftreten, wenn zwei konkrete Artefakte (Views) dieselbe Modellinformation an unterschiedlichen Stellen (z.B. in unterschiedlichen Dokumenten) jeweils inkonsistent zueinander beschreiben. So könnte z.B. in einem Dokument ein (globaler) Signaltyp stA mit einem formalen Parameter vom Datentyp Integer in einem anderen Dokument aber derselbe (globale) Signaltyp stA mit einem formalen Parameter vom Datentyp Real definiert werden. Da beim Parsen der Dokumente diejenigen konkreten Artefakte, welche dieselbe Modellinformation beschreiben, zu einem einzelnen abstrakten Artefakt

„abstrahiert“ werden (siehe Abschnitt 5.1.2 auf Seite 91), muss auch diese Art der Konsistenzverletzung im Parser geprüft werden.

Eine automatische Korrektur ist hier in manchen Fällen möglich, allerdings ist es in unseren Augen äußerst wichtig, dass der Entwickler dann vom Werkzeug über eine solche automatische Änderung benachrichtigt wird, weil sonst eventuelle Fehler verborgen bleiben könnten (vgl. Abschnitt 7.2.1 auf Seite 181).

8.1.2 Inkonsistenzen in der PROBAnD-Methode

Es muss nach obiger Argumentation also nur noch die Verletzung der statischen Semantik geprüft werden, da Verletzungen der abstrakten Syntax nicht möglich sind und Inkonsistenzen in der konkreten Syntax bereits durch den Syntax-Parser (erzeugt durch einen Parser-Generator, vgl. Abschnitt 2.1.2 auf Seite 13) behandelt werden.

Die beiden typischen Konsistenzprobleme, die es innerhalb des Parsers zu behandeln gilt, wurden oben bereits erläutert. Deren Detektion hängt sehr eng mit der konkreten Syntax der Dokumente zusammen. Sie ist zum Teil sehr aufwändig, da generische Lösungen nur schwer angewendet werden können. Hier steckt daher auch der größte Aufwand in der Werkzeugentwicklung wie wir in Abschnitt 10.2.3 auf Seite 329 bei dem Vergleich der Größen der implementierten Werkzeuge an Hand konkreter Zahlen zeigen werden. Auch der Einsatz unserer Aktionssprache AL++ würde diesen Aufwand nur marginal verringern. Wir werden daher keine konkreten Algorithmen für diesen Teil unserer Werkzeuge vorstellen, sondern uns hier auf die Verletzungen der statischen Semantik in den abstrakten Artefakten der PROBAnD-Methode konzentrieren.

Wie bereits diskutiert wurde, beschreiben die Multiplizitäten der Modelle einen Teil der Constraints, die es einzuhalten gilt. Diese Multiplizitäten beziehen sich dabei – wie bei der Einführung des Produktmodells bereits erwähnt wurde – immer auf die endgültige und vollständige Spezifikation. *Während* der Entwicklung muss als Untergrenze aller Multiplizitäts-Constraints daher auch die „0“ zugelassen werden, da ansonsten keine unvollständigen Modelle bearbeitet werden könnten.

Alle Einschränkungen des Modells lassen sich durch die Angabe von Multiplizitäten allerdings nicht beschreiben. Daher werden wir hier für das Produktmodell der PROBAnD-Methode diejenigen Constraints angeben, die bisher nicht explizit in unserem Produktmodell festgehalten wurden, sich aber aus der Semantik der Modellentitäten ergeben. Bei der Vorstellung dieser Constraints orientieren wir uns an der Struktur aus Abschnitt 6.2 auf Seite 148, weshalb die folgenden Unterüberschriften denen aus Kapitel 6 entsprechen.

Beschreibung der Anforderungen und der Objektstruktur

- $CA1$: „Jeder Need muss durch mindestens einen Task realisiert werden“. Wegen der Generalisierungsbeziehung lässt sich dies nicht durch eine „1..*“-Multiplizität an der Task-Seite der realizedBy-Relation ausdrücken, weil sonst auch jeder Task wieder durch mindestens einen weiteren Task realisiert werden müsste.
- $CA2$: „Alle ControlObjectTypes außer dem Wurzelobjekttyp müssen von einer Instantiation instanziiert werden“. Auch hier kann dies nicht durch einen Multiplizitäts-Constraint ausgedrückt werden, da die instantiates-Relation für den Wurzelknoten eben gerade nicht belegt sein darf.
- $CA3$: „Es dürfen keine zyklischen Instanziierungen definiert werden“. Im Detail bedeutet dieser Constraint, dass entlang eines Instanziierungspfades (von der Wurzel zu einem Blattknoten) ein Objekttyp immer nur einmal instanziiert werden darf. Da hier eine Abhängigkeit über mehrere Artefakte hinweg geprüft werden muss, lässt sich dieser Constraint nicht mit Multiplizitäten beschreiben und auch eine Formulierung durch einen logischen Ausdruck ist nicht unbedingt einfacht.

Statische Beschreibung von Objekteigenschaften und -kommunikation

- $CS1$: „Jedes Attribute muss von einer Strategy geschrieben werden, wenn es nicht eine defaultValue besitzt“ (siehe Constraint in Abschnitt 8.1.1)
- $CS2$: „Die formalen Parameter müssen linear nummeriert (geordnet) sein“. Insbesondere bedeutet dies, dass wenn ein FormalParameter mit einer parameterNbr von $n > 1$ existiert, auch ein formaler Parameter desselben Signaltyps mit einer parameterNbr von $n-1$ existieren muss. Desweiteren dürfen zwei Parameter desselben Signaltyps nicht dieselbe Nummer besitzen.
- $CS3$: „Jeder Signaltyp, der von einem Task eines Objekttyps produziert wird, muss von dem Task eines direkt benachbarten Objekttyps empfangen werden“. Eine solche Forderung ist notwendig, da die Kommunikation in der Baumstruktur der Spezifikation immer nur zwischen direkten Nachbarn erfolgen kann. Findet eine Kommunikation über mehrere Knoten des Baumes hinweg statt, so muss für jeden Zwischenknoten ein entsprechender Task definiert werden. Dieser Constraint ist sehr anspruchsvoll zu prüfen und über Multiplizitäten nicht zu beschreiben.

Operationale Verhaltensbeschreibung

- $CO1$: „Jeder Folgezustand muss mindestens einmal als Ausgangszustand auftauchen“. Diese Forderung ist notwendig, da die ControlObjectTypes in der

PROBAnD-Methode nie terminieren und daher kein Endzustand im Automaten existieren darf.

- $CO2$: „Jeder Ausgangszustand muss erreicht werden können und daher mindestens in einer Transition als Folgezustand definiert sein“. Ob der Zustand zur Laufzeit erreicht wird, ist dadurch natürlich noch nicht garantiert sondern müsste z. B. mit Hilfe einer Ereichbarkeitsanalyse (für SDL z. B. mit dem Telelogic Tau Validator [Dol03] oder allgemeiner mit einem „Model-Checker“ [Var04, 90f.]) geprüft werden.
- $CO3$: „Die Actions müssen geordnet sein“. Dies bedeutet, dass für die actionNbr zweier Aktionen (n_1 und n_2) derselben Transition stets gelten muss $n_1 > n_2$ oder $n_1 < n_2$, da sonst keine eindeutige Reihenfolge festzustellen ist.
- $CO4$: „Die aktuellen Parameter müssen linear nummeriert sein“ (siehe oben)
- $CO5$: „Die Anzahl der aktuellen Parameter von Signal muss mit der Anzahl der formalen Parameter von SignalType übereinstimmen“. Ist dies nicht gegeben, kann eine korrekte Wertbelegung der Parameter nicht erfolgen.
- $CO6$: „Ein Procedure muss entweder von einer LocalCallAction aufgerufen werden oder aber einen RemoteCallEvent auslösen“. Ansonsten macht die Definition einer solchen Prozedur keinen Sinn, da sie nicht verwendet wird.

Komplexe Artefakttypen

Die Konsistenzbedingungen für komplexe Artefakttypen lassen sich recht einfach und auch allgemein formulieren:

- $CKAi$: „Die Instanzen jedes abstrakten Artefakttyps A_i müssen von den jeweils relevanten komplexen Artefakten aggregiert werden“. Diese Bedingung ist Voraussetzung für den Einsatz des Unparsers zur Generierung von Dokumenten, da dessen Funktionsweise darauf beruht, dass das zu einem Dokument korrespondierende komplexe Artefakt alle atomaren Artefakte beinhaltet, zu denen in dem Dokument eine entsprechende View existiert (vgl. dazu auch Abschnitt 5.1.2 auf Seite 91).
- $CKKj$: „Von jedem komplexen Artefakttyp K_j muss mindestens eine Instanz existieren“. Diese Constraints müssen eingehalten werden, um alle Dokumenttypen erzeugen zu können. Es muss zumindest ein komplexes Artfakt K_j existieren, das keine aggregierten Artefakte besitzt. Dies entspricht in der konkreten Syntax einem leeren Dokument.

8.1.3 Inkonsistenzbehandlung im Werkzeug

Im Checker-Werkzeug werden die abstrakten Artefakte des Modells bzgl. ihrer „Well-formedness“ geprüft und eventuell korrigiert. Dabei werden zwei Techniken

zur Prüfung zum Einsatz gebracht: die Auswertung logischer Ausdrücke und die Implementierung von Prüfalgorithmen. Die zweite Art der Realisierung wurde eingesetzt, da es oftmals einfacher ist, eine algorithmische Prüfung von Constraints zur formulieren als deren deklarative Beschreibung (z.B. mittels logischer Ausdrücke) vorzunehmen (diese Erfahrung machten wir bereits bei der Gegenüberstellung von Modelltransformationsansätzen in Abschnitt 4.2.1 auf Seite 62).

Auswertung logischer Ausdrücke

Als technische Basis für die automatische Auswertung von Constraints, welche mit logischen Ausdrücken beschrieben werden, nutzten wir die von Hanke in seiner Diplomarbeit [Han03] realisierte Erweiterung des *Dresden OCL Toolkits* (siehe [Wie00] und [OCL04]). Dieses Werkzeug erlaubt die Anreicherung von Java-Klassen um OCL-Ausdrücke und deren Auswertung zur Laufzeit des Java-Programms.

Bei der *OCL* (*Object Constraint Language*, siehe [OMG03, Kapitel 6]) handelt es sich um ein formale Sprache, welche von der OMG zur exakten Spezifikation des UML-Metamodells eingesetzt wird. OCL-Ausdrücke können dabei für verschiedene Zwecke eingesetzt werden, unter anderem zur Spezifikation von Invarianten, zur Beschreibung von Vor- und Nachbedingungen und zur Modellierung von Guards an Zustandsübergängen (vgl. [OMG03, S.6-3]). Die für uns relevanten Constraints lassen sich durch die OCL-Invarianten beschreiben, deren Verletzung Inkonsistenzen im Modell anzeigen.

OCL-Ausdrücke sind dabei *immer* seiteneffektfrei, was impliziert, dass es nicht möglich ist, mit OCL Modelltransformationen *durchzuführen*. Die Spezifikation von Transformationen durch die Angabe von Vor- und Nachbedingungen ist hingegen möglich (vgl. [OMG04, S.5]) und entspricht damit dem in Abschnitt 4.2.1 auf Seite 62 eingeführten deklarativen Ansatz zur Modelltransformation, welcher vor der Ausführung zunächst eine Operationalisierung voraussetzt.

Die Konzepte der Object Constraint Language zur Beschreibung von Invarianten sind recht einfach. Zunächst muss der Kontext der Invariante angegeben werden, also das Artefakt, auf welches sich der Constraint bezieht. Dann folgt ein eindeutiger Name der Invariante, mit welcher die Art der Verletzung identifiziert werden kann. Zuletzt folgt der eigentliche logische Ausdruck zur Beschreibung der Invariante. Das obige Beispiel des Constraints $CS1$ für Attribute ließe sich mit der OCL wie folgt definieren:

```
context Attribute inv CS1:
    self.writingStrategy->isEmpty()
     implies self.value <> ''
```

Der Ausdruck `self` bezieht sich dabei auf die gerade evaluierte Instanz des Kontext-Artefakts.

Angenommen für das Attribut thetaSoll unseres *RoomAutomation*-Beispiels wäre nur eine lesende Strategie und nicht die Default-Belegung 20.0 wie in Tabelle 6-6 auf Seite 143 definiert. Eine Anwendung des Checker-Werkzeuges auf solch ein Modell würde dann die folgende Meldung liefern:

```
WARNUNG: Instanz 'thetaSoll' (Artefakttyp 'Attribute') verletzt
 Constraint 'CS1'
```

Das Dresden OCL Toolkit liefert dabei immer diejenigen Instanzen des Kontexttyps zurück, welche die jeweiligen Invarianten verletzen. Eine solche Kenntnis der Instanz reicht aber nicht immer aus, um den Fehler zu lokalisieren. So könnte das Attribut thetaSoll in mehreren Control-Object-Types definiert sein, weshalb man alle diese Objekttypen manuell untersuchen müsste, um die fehlerhafte Stelle zu finden. Mit Hilfe des generischen Link-Zugriffs von AL++ ließe sich die „Einbettung" der Instanz allerdings feststellen und zusätzlich ausgeben. Für das obige Beispiel erhielte man die Auflistung aller Links zu:

```
readingStrategy = T7Strategy
itsDataType = Real
```

Mit Hilfe der Angabe der Strategie kann man nun das Attribut eindeutig über den Task T7 zu dem Objekttyp TempCtrl zurückverfolgen. Diese Verfolgbarkeit wird von einem Browser-Werkzeug, welches wir in Abschnitt 8.4.1 auf Seite 232 vorstellen werden, unterstützt.

So wie für das obige Beispiel der formalen Derstellung von Constraint $CS1$ in der OCL, kann man nun versuchen auch die verbleibenden Constraints aus Abschnitt 8.1.2 in OCL-Ausdrücke zu überführen und damit einer automatischen Überprüfung zugänglich zu machen. Im Folgenden führen wir einige dieser Constraints auf, für welche eine solche Formalisierung einfach gelingt.

Zunächst die OCL-Formulierung von Constraint $CA1$:

```
context Requirement inv CA1:
    self.oclIsTypeOf(Need)
     implies self.realizingTask->notEmpty()
```

Hier wird zunächst festgestellt, ob es sich bei der Requirement-Instanz um die Instanz eines Needs handelt (`oclIsTypeOf()`). Dann muss diese Instanz mindestens einen realizing-Link zu einem Task besitzen.

Sehr ähnlich ist auch die Realisierung von $CA2$:

```
context ControlObjectType inv CA2:
    self.itsObjectStructure->isEmpty()
     implies self.instantiatingInstantiation->notEmpty()
```

Mittels des Rollennamens itsObjectStructure wird auf die Instanz am Ende der topLevelControlObjectType-Relation des komplexen Artefakts ObjectStructure zugegriffen. Findet sich dort keine Instanz, dann handelt es sich bei dem aktuell betrachteten Objekttyp (`self`) nicht um den Wurzelknoten. In einem solchen Fall muss dann dieser Objekttyp irgendwo instanziiert werden (ein instantiates-Link muss existieren).

Um einiges komplizierter ist der OCL-Constraint für $CS2$:

```
context SignalType inv CS2:
    self.itsParameter->forAll(fp1 : Parameter |
     let n1 : Integer = fp1.parameterNbr in
     if n1 > 0 then
        self.itsParameter->select(fp2 : Parameter |
         fp2.parameterNbr = n1-1)->notEmpty()
        and self.itsParameter->select(fp2 : Parameter |
         fp2.parameterNbr = n1)->size() = 1
    else
        true
    endif)
```

Das `forAll`-Schlüsselwort steht für den $\forall$-Quantor, weshalb für alle Parameter `fp1` gelten muss, dass wenn der Wert des Attributes `parameterNbr` (mit Hilfe des `let`-Ausdrucks der Variable `n1` zugewiesen) größer Null ist, ein Parameter existieren muss, der eine `parameterNbr` von `n1-1` besitzt. Der `select`-Ausdruck wählt aus der angegebenen Menge diejenigen Elementen aus, welche die in Klammern angegebene Bedingung erfüllen. Ist diese Menge nichtleer, so wurde mindestens ein gewünschter Parameter gefunden.

Neben der Existenz eines Parameters mit `n1-1` wird zusätzlich geprüft, ob der Parameter mit der Nummer `n1` wirklich nur einmal für den Signaltyp existiert, also ob die von `select()` gelieferte Menge die Größe eins hat.

Wie man an diesem letzten Beispiel sieht, können solche Ausdrücke sehr komplex und verschachtelt sein. Sourrouille et al. gehen in [SoC02, S.14] sogar soweit zu sagen, dass „[...] einige Regeln nicht in OCL ausgedrückt werden können [und] OCL nicht einfach anzuwenden ist [...]“. Ich bin der Meinung, dass es oftmals einfacher ist, eine algorithmische Lösung zur Überprüfung von Constraints anzugeben, als diese in OCL zu formulieren. Für einen solchen Fall wird im folgenden Abschnitt ein Beispiel vorgestellt.

Implementierung von Prüfalgorithmen

Die Aktionssprache AL++ bietet mit den Möglichkeiten beliebige temporäre Datenstrukturen verwenden und Algorithmen rekursiv beschreiben zu können, oft die elegantere aber dennoch kompaktere Lösung im Vergleich zur OCL.

Am Beispiel von Constraint $CA3$, welcher zyklische Instanziierungen verbietet, soll dies illustriert werden. Die Strategie der Detektion basiert dabei darauf, alle möglichen Pfade vom Wurzelknoten zu jedem Blattknoten zu durchlaufen und zu prüfen, ob auf einem solchen Pfad ein ControlObjectType mehr als einmal instanziiert wird. Ausgangspunkt für die dazu verwendete Rekursion ist der komplexe Artefakttyp ObjectStructure, wo die Operation `checkCA3()` definiert wird:

```
1   /* ObjectStructure: */
2   public void checkCA3() {
3       topLevelControlObjectType.checkCA3(new HashSet(), "");
4   }
```

Hier beginnt die Rekursion bei der Wurzel des Instanziierungsbaums mit der Operation `checkAC3()` des Artefakttyps ControlObjectType:

```
1   /* ControlObjectType: */
2   public void checkCA3(Set visitedCots, String instName) {
3       if(visitedCots.contains(this)) {
4           System.err.println("WARNUNG: Instanz '"+instName+
5            "' (Objekttyp '"+name+"') verletzt Constraint 'CA3'");
6           return;
7       } else {
8           visitedCots.add(this);
9       }
10      ControlObjectType aggregatedCot;
11      foreach(inst in aggregatedInstantiation) {
12          aggregatedCot = inst.instantiatedControlObjectType;
13          aggregatedCot.checkCA3(new HashSet(visitedCots),
14           inst.name+"."+instName);
15      }
16  }
```

In den Zeilen 3 bis 7 findet die eigentliche Feststellung einer Verletzung statt. Hier wird geprüft, ob der aktuelle Objekttyp bereits auf dem bisherigen Pfad (`visitedCots`) instanziiert wurde. Ist dies der Fall, wird eine entsprechende Fehlermeldung mit der Angabe des Instanzennamens (`instName`), der die Stelle der falschen Instanziierung beschreibt, ausgegeben.

In der zweiten Hälfte des Algorithmus wird der aktuelle Objekttyp in die Menge der schon besuchten Control-Object-Types aufgenommen und weiter im Instanziie-

rungsbaum abgestiegen. Dabei erhält jeder neue Zweig eine Kopie der bis dahin berechneten Menge von Objekttypen (Zeile 13). Der `instName` entsteht durch eine Konkatenation des Namens der Instantiation. Damit entstehen Instanzennamen in der Form wie sie bereits in Abschnitt 6.1.3 auf Seite 138 beschrieben wurden.

Angenommen, in unserem *RoomAutomation*-Beispiel würde der TempSens Objekttyp fälschlicherweise eine tmc1-Instanz vom Type TempCtrl aggregieren. Dann würden wir die folgende Verletzungsmeldung erhalten:

```
WARNUNG: Instanz 'tmc1.its1.tmc1.obj1' (Objekttyp 'TempCtrl') verletzt
 Constraint 'CA3'
```

Dieses Beispiel soll an dieser Stelle zur Illustration genügen. In Abschnitt 8.2 auf Seite 208 werden wir diese Technik nochmals nutzen, um sehr komplexe Analysen durchzuführen. Insbesondere werden wir dort in Abschnitt 8.2.1 einen anspruchsvollen Algorithmus präsentieren, den man in leicht modifizierter Form für die Überprüfung des $CS3$-Constraints einsetzen könnte.

Automatische Ableitung von Invarianten für Multiplizitäten

Wie wir bereits in Abschnitt 8.1.1 angedeutet hatten, gibt es zwei Klassen von Multiplizitäten, deren Einhaltung wir auf der Basis unserer Aktionssprache nicht garantieren können.

In die erste Klasse fallen Relationen mit einer Multiplizität von „1“, was impliziert, dass das Assoziationsende immer belegt sein muss. Dies wird in unserer Sprache nicht erzwungen, da eine Belegung des Link-Endes mit `null` legitim ist. In die zweite Klassen fallen alle Multiplizitäten „$m..n$“ mit $n > 1$, was die Interpretation eines solchen Assoziationsendes als eine Menge von Instanzen impliziert. Ein Forcieren der Größe einer solchen Menge ist in unserem Ansatzes zunächst nicht vorgesehen.

Aus obigen Gründen ist es angebracht, diese Constraints durch OCL-Invarianten zu beschreiben. Dies gelingt recht einfach durch die Spezifikation eines logischen Ausdrucks der folgenden Form (`<ContextType>`, `<roleName>`, m und n sind dabei jeweils entsprechend zu ersetzen):

```
context <ContextType> inv C_<ContextType>_<roleName>:
    self.<roleName>->size() >= m
    and
    self.<roleName>->size() <= n
```

Handelt es sich bei n um das Literal „*“, so sind die beiden letzten Zeilen entsprechend wegzulassen. Handelt es sich um $m = 0$, kann die zweite Zeile ausgelassen werden. Gilt $m = n$, so kann man das Constraint auf

`self.<roleName>->size()` = m

reduzieren.

Das obige Muster muss man nun nur noch für alle Artefakttypen und Relationen des Produktmodells anwenden. Da es sich hierbei um eine sehr reguläre Aktivität handelt, bietet sich natürlich wieder eine Automatisierung an.

Hanke stellt dazu in [Han03] eine Lösung auf der Basis des Austauschformats XMI (vgl. Abschnitt 5.3.1 auf Seite 118) und der Transformationssprache XSLT (vgl. Abschnitt 4.2.2 auf Seite 65) vor. Diese XSLT-Lösung benötigt 85 Zeilen und beinhaltet zum Teil sehr komplexe „Pfadausdrücke“ von bis zu 180 Zeichen Länge zur Modellnavigation. Wir wollen dieser Lösung eine Alternative in der Sprache AL++ gegenüberstellen, die von der Generizität der Sprache und der Modelle (Produktmetamodell) profitiert.

Hier zeigt sich wieder einmal die Mächtigkeit des Metamodellansatzes. Denn so wie wir bisher Modelltransformationen und -Analysen auf der Ebene M_1 durchführten, lassen sich dieselben Konzepte auch für eine Modellanalyse auf Ebene M_2 einsetzen. Damit können wir unser Produktmodell (M_2) analysieren und die gewünschten Multiplizitäts-Invarianten ableiten.

Dazu beginnen wir mit der Definition der generischen Methode `genConstr()` im Elementtyp ArtefactType, welcher die Oberklasse aller Elementtypen des Produktmetamodells darstellt (vgl. Abschnitt 5.1.1 auf Seite 86).

```
1   /* ArtefactType: */
2   private Set artefactTypesToBeVisited = new HashSet();
3   private Set visitedArtefactTypes = new HashSet();
4   private Writer out;
5
6   public void genConstr(Writer out, ArtefactType root) {
7       this.out = out;
8       artefactTypesToBeVisited.add(root);
9       ArtefactType artefactType;
10      while(!artefactTypesToBeVisited.isEmpty()) {
11          foreach(artefactType in artefactTypesToBeVisited) {
12              genConstrForArtefactType(artefactType);
13              break;
14          }
15          visitedArtefactTypes.add(artefactType);
16          artefactTypesToBeVisited.remove(artefactType);
17      }
18  }
```

Dem Leser wird die Ähnlichkeit zu dem in Abschnitt 5.2.4 auf Seite 109 vorgestellten Algorithmus für das Marshalling von M_1-Modellen auffallen. Analog zu diesem Algorithmus laufen wir auch hier über alle Elemente einer Menge `artefactTypesToBeVisited` und erzeugen die entsprechenden Constraints (Zeile 12), welche in eine über `out` spezifizierte Datei geschrieben werden.

Begonnen wird mit einem initialen Wurzelknoten (`root`), von dem aus der gesamte Modellgraph durchlaufen wird. Wie auch beim Marshalling ist die Voraussetzung für die Funktionsfähigkeit des Algorithmus, dass das analysierte Modell einen zusammenhängenden Graphen darstellt. Für das Produktmodell der PROBAnD-Methode ist dies garantiert, da ausgehend von dem komplexen Artefakttyp RequirementsSpecification alle verbleibenden Artefakttypen in einer hierarchischen Weise aggregiert werden (vgl. Abschnitt 6.2.4 auf Seite 165).

Für jeden Artefakttyp müssen wir nun weiter dessen Relationen bestimmen und für jede dieser Relationen die entsprechenden Multiplizitäten.

```
1   /* ArtefactType: */
2   private void genConstrForArtefactType(
3    ArtefactType artefactType) {
4       genConstrForAssociationType(artefactType, "AssociationType", true);
5       genConstrForAssociationType(artefactType, "AggregationType", true);
6       if(artefactType instanceof AtomicArtefactType) {
7           genConstrForAssociationType(artefactType,
8            "GeneralizationType", false);
9       }
10  }
```

Außer für allgemeine Assoziationen müssen die Constraints auch für Aggregationen berechnet werden. Desweiteren muss der Modellgraph für atomare Artefakttypen auch entlang der Generalisierungsrelation verfolgt werden, da generellere Artefakttypen (wie z.B. Requirement) oder speziellere Artefakttypen (wie z.B. RemoteCallEvent) nicht unbedingt über eine Aggregations- oder Assoziationsbeziehung mit dem Rest des M_2-Modells verbunden sind.

Wir definieren dazu die generische Operation `genConstrForAssociationType()`, welche in Abhängigkeit des Namens des Relationstyps diese Aufgabe erledigt. Anders als bei den Relationstypen AssociationType und AggregationType macht es bei GeneralizationType allerdings keinen Sinn, einen Multiplizitäts-Constraint zu definieren, da dieser nicht auf der Ebene der Modellinstanzen sichtbar ist (die Generalisierung besteht immer nur zwischen Typen und nicht zwischen Instanzen). Dies kennzeichnen wir durch die Übergabe des Parameters `false` an die Operation `genConstrForAssociationType()`:

```
1   /* ArtefactType: */
```

```
2   private void genConstrForAssociationType(ArtefactType artefactType,
3    String relationTypeName, boolean createConstr) {
4       ArtefactType destArtefactType;
5       foreach(String roleName in artefactType.#relationTypeName) {
6           Multiplicity mult;
7           (null, null, mult, null, destArtefactType) =
8            (artefactType.#relationTypeName, roleName);
```

Mittels des Ausdrucks `artefactType.#relationTypeName` bestimmt man zunächst alle Rollennamen und die jeweiligen Multiplizitäten. Im Produktmodell der PROBAnD-Methode würde man für den Relationstyp AssociationType ausgehend vom Artefakttyp Task eine Menge mit den Einträgen `"realizedRequirement"`, `"realizingStrategy"`, und `"implementingControlObjectType"` erhalten.

```
9           if(createConstr) {
10              out.println("context "+artefactType.name+
11               " inv C_"+artefactType.name+"_"+roleName+":");
12              if(mult.lower == mult.upper) {
13                  out.println("self."+roleName"->size() = "+mult.lower);
14              } else {
15                  if(mult.lower != 0)
16                      out.println("self."+roleName+
17                       "->size() >= "+mult.lower);
18                  if(mult.upper != *) {
19                      if(mult.lower != 0)
20                          out.println("and");
21                      out.println("self."+roleName+
22                       "->size() <= "+mult.upper);
23                  }
24              }
25          }
26          if(!visitedArtefactTypes.contains(destArtefactType))
27              artefactTypesToBeVisited.add(destArtefactType);
28      }
29  }
```

Mit den Relationsinformationen ist die Generierung der eigentlichen Constraints nun sehr einfach. Unter Berücksichtigung der oben erläuterten Fälle erfolgt dies in den Zeilen 9 bis 25. Als Letztes müssen nun nur noch alle über die Relationen verbundenen Artefakttypen in die Menge der zu untersuchenden Elemente aufgenommen werden, falls dies nicht schon geschehen ist.

Wendet man die `genConstr()`-Operation auf die RequirementsSpecification-Instanz der PROBAnD-Methode an, dann erhält man für das obige Beispiel die folgenden Constraints:

```
context Task inv C_Task_realizedRequirement:
    self.realizedRequirement->size() > 1

context Task inv C_Task_realizingStrategy:
    self.realizingStrategy->size() = 1

context Task inv C_Task_implementingControlObjectType:
    self.implementingControlObjectType->size() = 1
```

Hiermit erzielten wir wieder eine beachtliche Reduktion des Aufwandes zur Automatisierung von Aktivitäten (57 Zeilen der obigen Lösung im Vgl. zu 85 Zeilen der XSLT-Lösung). Dies unterstreicht die Mächtigkeit und den Abstraktionsgewinn unserer Aktionssprache AL++. Desweiteren erlaubt diese Sprache eine Modelltransformation auf allen Ebenen der Metamodellhierarchie, weshalb ein Entwickler prinzipiell nur *eine* solche Sprache beherrschen muss.

Durch eine geringe Modifikation des obigen Algorithmus ließe sich dann auch das zu Beginn von Abschnitt 8.1.2 aufgeführte Problem, dass während der Entwicklung prinzipiell immer eine Untergrenze von „0“ für eine Multiplizität gilt, lösen. Man würde einfach eine getrennte Liste von Constraints erzeugen und diese modifizierten Constraints prüfen.

Alternativ zu diesem indirekten Weg der Multiplizitätsprüfung über die OCL-Constraints kann man diese Prüfung natürlich auch direkt über einen generischen AL++-Prüfalgorithmus, der die Modellebenen M_1, M_2 und M_3 gleichzeitig berücksichtigt, realisieren.

Korrektur von Konsistenzverletzungen

Wurde eine Inkonsistenz in den abstrakten Artefakten festgestellt, so ist es angebracht diese zu korrigieren. Liegt genügend Entwicklungsinformation vor, kann man eine solche Inkonsistenz prinzipiell auch automatisch korrigieren. Allerdings sollte man hier stets Vorsicht walten lassen, um keine Fehler zu überdecken (siehe dazu auch die Diskussion in Abschnitt 7.2.1 auf Seite 181 und Abschnitt 10.3.1 auf Seite 336).

Die Verletzung von Multiplizitäten und der Constraints aus Abschnitt 8.1.2 lassen sich in der PROBAnD-Methode mit Ausnahme von $CKAi$ und $CKKj$ nicht automatisch korrigieren, weil die zur Verfügung stehende Information zu gering ist. Wie sollte man z.B. im Falle von $CS1$ beurteilen, ob man einen Default-Wert für das Attribut anlegen soll oder eine produzierende Strategy erzeugen muss?

Die automatische Korrektur der $CKAi$- und $CKKj$-Constraints kann allerdings generisch mit AL++ beschrieben werden. Die Korrektur der Inkonsistenzen wird hierbei zusammen mit deren Detektion in den folgenden Operationen realisiert.

Dabei stellt `checkCK()` des komplexen Artefakttyps RequirementsSpecification den Einstiegspunkt dar:

```
1   /* RequirementsSpecification: */
2   Map allInstances;
3   public void checkCK() {
4       allInstances = this.collectAllInstances();
5       checkCKComplexArtefact(Needs);
6       checkCKComplexArtefact(TaskList);
7       checkCKComplexArtefact(ObjectStructure);
8       checkCKComplexArtefact(System);
9       checkCKCotc();
10  }
```

Zunächst wird eine Tabelle aller Artefaktinstanzen der aktuellen RequirementsSpecification-Instanz (`this`) erzeugt. Der Algorithmus ist analog zu dem Marshaller-Algorithmus, den wir bereits weiter oben in ähnlicher Weise zur Ableitung der OCL-Constraints, eingesetzt hatten. Auch `collectAllInstances()` läuft über alle Instanzen nur dass die Instanzen diesmal an Hand ihres Typs (als Schlüssel) in eine Tabelle (`Map`) eingetragen werden. Damit ist es in den folgenden Operationen möglich, durch nur einen Zugriff auf diese Tabelle alle Instanzen eines bestimmten Typs zu erhalten.

Betrachtet man sich die Struktur der komplexen Artefakttypen der PROBAnD-Methode (siehe Abschnitt 6.2.4 auf Seite 165), so stellt man fest, dass alle Typen außer der ControlObjectTypeConfiguration dieselbe reguläre Struktur aufweisen und auch stets alle Instanzen der aggregierten Typen referenzieren müssen. Dies bietet uns die Möglichkeit mit einer einzigen generischen Operation `checkCKcomplexArtefact()` der RequirementsSpecification alle diese Artefakte zu prüfen:

```
1   /* RequirementsSpecification: */
2   public void checkCKComplexArtefact(ComplexArtefactType cat) {
3       String roleName = "its"+cat.name;
4       ComplexArtefactType.anyinstance ca = this.#roleName;
5       if(ca == null) {
6           System.err.println("WARNUNG: Komplexes Artefakt '"+cat.name+
7            "' verletzt Constraint CKKj, neue Instanz wird erzeugt.");
8           ca = new #(cat.name)(cat.name);
9           this.#roleName = ca;
10      }
```

Die Überprüfung der Constraints $CKKj$ erfolgt zuerst. Über den generischen Zugriff auf die Aggregationsrelation der RequirementsSpecification besorgt man sich zunächst die aggregierte Instanz des spezifizierten ComplexArtefactType `cat`. Ist der

Link nicht belegt, so handelt es sich um eine Constraint-Verletzung und als Korrekturmaßnahme wird eine neue Instanz (mit dem Namen des Typen) angelegt und aggregiert.

Im weiteren Verlauf der `checkCKComplexArtefact()`-Operation werden die Constraints $CKAi$ geprüft:

```
11      Set artefactsInSpecification;
12      Set artefactsInComplexArtefact;
13      foreach(String roleName in cat.part) {
14          ArtefactType artefactType;
15          (null, null, null, null, null, artefactType) = (cat.part,
16           roleName);
17          artefactsInSpecification = allInstances.get(artefactType);
18          artefactsInComplexArtefact = ca.#roleName;
19          artefactsInSpecification.
20           removeAll(artefactsInComplexArtefact);
21          foreach(ArtefactType.anyinstance artefact in
22           artefactsInSpecification) {
23              System.err.println("WARNUNG: Artefakt '"+artefact.name+
24               "'(Typ '"+artefact.type.name+
25               "') verletzt Constraint CKAi, wird hinzugefuegt.");
26              ca.#roleName += artefact;
27          }
28      }
29  }
```

Für alle vom dem betrachteten komplexen Artefakttyp aggregierten Artefakttypen besorgt man sich zunächst die in der gesamten Spezifikation gefundenen Instanzen (diese stehen in der `allInstances`-Tabelle). Da es sich bei allen Agregationsbeziehungen der komplexen Artefakte der PROBAnD-Methode um eine zu-n-Beziehung handelt, erhält man die Menge der von dem betrachteten komplexen Artefakt aggregierten Instanzen einfach über den entsprechenden Link-Zugriff (Zeile 18).

Diejenigen Instanzen, die noch nicht vom komplexen Artefakt aggregiert werden, werden über die Mengendifferenz (`removeAll()`-Methode) ermittelt. Alle in der Menge verbleibenden Instanzen werden dann hinzugefügt und die Constraint-Verletzung jeweils ausgegeben (Zeilen 21 bis 27).

Wie schon erwähnt, ist diese Vorgehensweise für die ControlObjectTypeConfiguration nicht so einfach zu realisieren. Insbesondere weil in einem solchen komplexen Artefakt immer nur solche Artefakte aggregiert werden dürfen, welche sich auf den jeweiligen Objekttyp beziehen.

Einen solchen Fall festzustellen erfordert alle Relationen, welche den aggregierten Artefakttyp mit dem Artefakttyp ControlObjectType verbinden, zu verfolgen. So dürfen in einer ControlObjectTypeConfiguration des Control-Object-Types TempCtrl nur diejenigen Attribute aggregiert werden, welche über reads-/writes-Links mit einer Strategie verbunden sind, deren realisierter Task von dem Objekttyp TempCtrl implementiert wird. Insbesondere ist es notwendig diesen „Relationspfad“ (und damit die Richtung der „Suche“) exakt anzugeben. Liefe man allgemein über alle Links, so würde man immer eine Verbindung zwischen Artefakt und Objekttyp feststellen, da in unseren Modellgraphen immer ein Pfad zwischen zwei beliebigen Modellelementen existiert.

Da sich die Algorithmen außer dieser zusätzlichen Prüfung nur marginal unterscheiden werden wir nur den Algorithmus zur Prüfung der $CKAi$-Constraints vorstellen. Dazu wird zunächst in der ControlObjectTypeConfiguration die Operation `checkCKACotc()` definiert.

```
1   /* ControlObjectTypeConfiguration: */
2   public void checkCKACotc(ArtefactType artefactType,
3    String[] path, Map allInstances) {
4       ControlObjectType cot = this.itsControlObjectType;
5       Set artefactsInSpecification = allInstances.get(artefactType);
6       String roleName = "its"+artefactType.name;
7       Set artefactsInComplexArtefact = this.#roleName;
8       artefactsInSpecification.removeAll(artefactsInComplexArtefact);
9       foreach(ArtefactType.anyinstance artefact in
10       artefactsInSpecification) {
11          if(artefact.isConnectedVia(path, 0, cot)) {
12              System.err.println("WARNUNG: Artefakt '"+artefact.name+
13               "'(Typ '"+artefact.type.name+
14               "') verletzt Constraint CKAi, wird hinzugefuegt.");
15              this.#roleName += artefact;
16          }
17      }
18  }
```

Diese Operation unterscheidet sich nur in den Zeilen 4 und 11 von dem entsprechenden Abschnitt in der generischen Operation `checkCKComplexArtefact` im komplexen Artefakttyp RequirementsSpecification. Nur wenn das Artefakt `artefact` über den angegebenen Pfad von Relationen wirklich mit der Instanz des betrachteten ControlObjectTypes (`cot`) verbunden ist, wird `artefact` hinzugefügt. Als Parameter erwartet die `isConnectedVia()`-Operation die Rollennamen als Array von Strings.

Damit kann man nun also wieder mit Hilfe einer generischen Operation alle relevanten Fälle prüfen. Die folgende Operation `checkCKCotc()` zeigt beispielhaft den oben beschriebenen Fall der Verbindung des Artefakttyps Attribute:

```
1   /* RequirementsSpecification: */
2   public void checkCKCotc() {
3       /* checkCKKCotc - analog zu oben */
4
5       String[] path = new String[] {"writingStrategy", "realizedTask",
6        "implementingControlObjectType"};
7       checkCKACotc(Attribute, path, allInstances);
8       path = new String[] {"readingStrategy", "realizedTask",
9        "implementingControlObjectType"}
10      checkCKACotc(Attribute, path, allInstances);
11      /* ... */
12  }
```

Die Spezifikation von `isConnectedVia()` erfolgt dabei als eine Operation der Potenz 2 von ArtefactType, d. h. dass die eigentlichen Operationen der Potenz 1 in allen Instanzen des ArtefactTypes existieren:

```
1   /* ArtefactType: */
2   public void isConnectedVia^2_2(String[] path, int pos,
3    ArtefactType.anyinstance targetArtefact) {
4       boolean returnVal = false;
5       if(pos == path.length) {
6           if(this == targetArtefact)
7               returnVal = true;
8       } else {
9           foreach(ArtefactType.anyinstance artefact in this.#path[pos]) {
10              returnVal = returnVal |
11               artefact.isConnectedVia(path, pos+1, targetArtefact);
12          }
13      }
14      return returnVal;
15  }
```

Zunächst wird geprüft, ob wir bereits am Ende des Pfades angekommen sind, d. h. ob die aktuell geprüfte Stelle im Pfad-Array der Länge des Arrays entspricht. Handelt es sich dann bei dem Artefakt um das gesuchte Zielartefakt, so wird die Rekursion beendet und `true` zurückgeliefert. Ansonsten läuft man weiter über alle Instanzen, die über den Rollennamen an der aktuellen Stelle in `path` referenziert werden. Die Operation `isConnectedVia()` liefert immer dann `true`, wenn ein Weg

von der Start- zur Zielinstanz gefunden wurde (Oder-Verknüpfung durch „| “ in Zeile 10).

Auch für diese Anwendung zeigen sich also wieder die Vorzüge der verbesserten Multiebenenmodellierung und der Aktionssprache AL++.

8.2 Detektion von Feature-Interaktionen

Viele der obigen Inkonsistenzen ließen sich relativ einfach durch die Überprüfung simpler „Well-formedness“ feststellen. Auch eine syntaktische Prüfung während des Parse-Vorgangs ist verhältnismäßig leicht zu realisieren. In diesem Abschnitt werden wir demgegenüber eine anspruchsvolle statische Analyse der PROBAnD-Artefakte vorstellen. Zur Realisierung dieser komplexen Analyse machen wir speziell von der Mächtigkeit unserer Aktionssprache bzgl. Rekursion Gebrauch.

Das Feature-Interaktionsproblem

Betrachtet man die Evolution eines Software-Systems, so können insbesondere zwei kritische Probleme auftauchen. Eines dieser Probleme tritt bei der Erweiterung des Systems zu Tage, also dann wenn zusätzliche Produktmerkmale (Features) aufgenommen werden, um neue Benutzeranforderungen zu realisieren. Neben dem erwünschten Verhalten kann die neue Funktionalität auch *unerwünschte* Wechselwirkungen mit den alten Teilen des Systems aufweisen, weshalb ein fehlerhaftes Systemverhalten resultieren kann.

Das zweite Problem tritt bei der Wiederverwendung eines existierenden Systems auf. Werden nur Teile des Gesamtsystems wiederverwendet, so müssen *notwendige* Wechselwirkungen mit alten Systemteilen berücksichtigt werden, weil sonst die wiederverwendeten Komponenten nicht funktionsfähig sind.

In der Domäne der Telekommunikationssysteme ist dieses Problem der *Feature-Interaktion* seit längerer Zeit bekannt (siehe z. B. den Übersichtsartikel von Calder et al. [CKM03]) und wurde dort entsprechend bearbeitet, wie man z. B. an Veranstaltungen wie dem regelmäßig stattfindenden „Feature Interaction Workshop“ [AmL03] sieht. Für den Bereich der Mess-, Steuer- und Regelsysteme (MSR-Systeme) wurde diese Problemklasse bisher allerdings nicht systematisch untersucht, was damit zu begründen ist, dass die Komplexität solcher Systeme erst in jüngster Zeit kritische Ausmaße annimmt (siehe auch die Einleitung zu dieser Arbeit).

Als kennzeichnender Unterschied zu den Telekommunikationssystemen sind solche Systeme immer in eine physikalische Umgebung eingebettet, die nicht Teil des jeweiligen Software-Systems ist (siehe Abschnitt 6.1.1 auf Seite 132). Wegen der besonderen Rolle dieser Umgebung genügt es daher nicht, Feature-Interaktionen nur *innerhalb* des Software-Systems zu behandeln, sondern auch Wechselwirkungen, welche durch die Umgebung entstehen, müssen zwingend berücksichtigt werden.

Bereits in unserem kleinen *RoomAutomation*-Beispiel aus Kapitel 6 findet sich eine solche Interaktion, welche durch die Umgebung ausgelöst wird. Wir hatten in diesem Beispiel die Licht- und Temperaturregelung völlig unabhängig voneinander realisiert (es besteht keine realizedBy-Beziehung zwischen den Anforderungen N1 bis N3 und der Anforderung N4). Da wir jedoch Sonnenlicht als natürliche Lichtquelle zulassen (dies realisiert die Anforderung des „Energiesparens"), kann durch die Wärmeenergie der Sonne der Raum aufgeheizt werden. Damit entsteht eine unerwünschte Interaktion zwischen der Licht- und der Temperaturregelung, denn die Temperaturregelung hat in einer solchen Situation keine Möglichkeit, das Aufheizen des Raums zu verhindern.

Bevor man nun Feature-Interaktionen überhaupt behandeln kann, muss eine Aktivität vorausgehen, welche solche Interaktionen detektiert. Wir werden in diesem Abschnitt zeigen, wie eine solche Detektion für die Dokumente der PROBAnD-Methode automatisiert werden kann. Eine solche Automatisierung ist unabdingbar, will man von einer Qualitätssicherung durch die Betrachtung von Feature-Interaktionen überhaupt profitieren, da eine manuelle Detektion nicht realistisch ist. Die Komplexität einer solchen Interaktionsdetektion werden wir in Abschnitt 8.2.3 auf Seite 226 illustrieren.

Die Konzepte dieses Ansatzes und die Ergebnisse verschiedener Fallstudien werden sehr detailliert in dem gemeinsam mit Webel veröffentlichten Tagungsbeitrag [MeW03] und in dem Zeitschriftenaufsatz [Met04] dargelegt. Wir werden hier daher nicht alle dort vorgestellten Konzepte in demselben Detaillierungsgrad wiederholen, sondern unsere Aufmerksamkeit auf die konkrete Formulierung der Detektionsalgorithmen (beschrieben mit AL++) richten.

8.2.1 Detektionsalgorithmen

Die Anforderungen (Needs) an ein Produkt manifestieren sich in dessen beobachtbaren Eigenschaften, also den Features. Daher sind Feature-Interaktionen bereits in den Anforderungsdokumenten zu finden (vgl. [Bre01]) und lassen sich folglich in den von uns betrachteten Systemen durch die Analyse der Abhängigkeiten zwischen den Requirements feststellen. Dazu wird von den feingranularen Verfolgbarkeitsrelationen zwischen den atomaren Artefakten der PROBAnD-Methode Gebrauch gemacht (siehe Abschnitt 6.1.4 auf Seite 147).

Je nach Entwicklungsfortschritt stehen mehr oder weniger Daten (atomare Artefakte und Links) für die Analyse bzgl. der Feature-Interaktionen zu Verfügung. Es sind daher verschiedene Ebenen der Analyse in Abhängigkeit von der betrachteten Information möglich, wobei die Analyseergebnisse mit jeder weiteren Ebene verfeinert werden.

Detektion auf Anforderungsebene

Zu Beginn einer Entwicklung mit der PROBAnD-Methode sind Needs und Tasks die einzigen atomaren Artefakte, die spezifiziert wurden. Die Interaktionsdetektion auf dieser Ebene kann also nur auf Basis dieser Artefakte zusammen mit der realizedBy-Relation zwischen diesen erfolgen. Diese Artefakte beschreiben einen Abhängigkeitsgraph, in welchem wir sog. *Interaktionspunkte* (engl. „*points of interaction*“) identifizieren können. Ein Interaktionspunkt ist dabei ein Knoten, welcher mehr als einen Need realisiert und mehr als einen direkten Elternknoten besitzt. Ausgehend von solchen Interaktionspunkten lassen sich dann die eigentlichen Feature-Interaktionen ableiten.

In Abb. 8-2 ist ein solcher Abhängigkeitsgraph für das *RoomAutomation*-Beispiel aus Abschnitt 6.1.3 auf Seite 138 gezeigt.

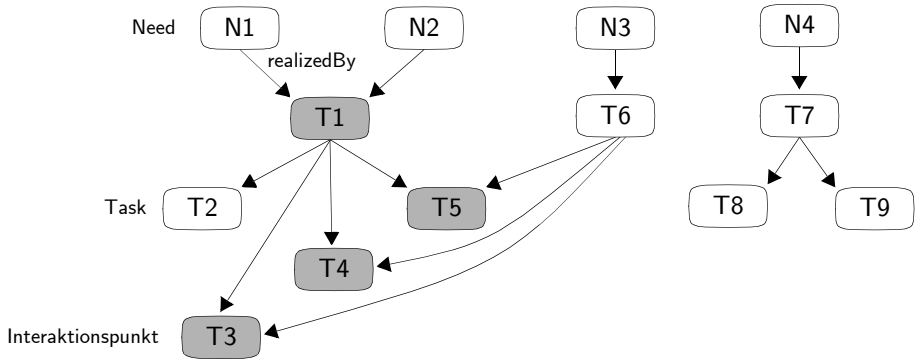


Abbildung 8-2. Abhängigkeitsgraph für das *RoomAutomation*-Beispiel

Wie man erkennt, wurden vier Interaktionspunkte (T1, T3, T4 und T5) identifiziert. Ausgehend von diesen Interaktionspunkten lassen sich Feature-Interaktionen zwischen N1 und N2 (im Punkt T1) und zwischen N1, N2 und N3 (in den Punkten T3, T4 und T5) ableiten.

Die algorithmische Detektion der Interaktionspunkte verläuft wie folgt:

1. Für jeden Task T_i $(i = 1, \ldots, n)$ wird eine leere Menge S_i angelegt.
2. Ausgehend von jedem Need N_j $(j = 1, \ldots, m)$ verfolgt man alle realizedBy-Relationen von diesem Need zu allen Tasks T_k, welche diesen Need realisieren und fügt N_j zu der Menge S_k für jeden passierten Task T_k hinzu.
3. Ein Task T_i kann als Interaktionspunkt identifiziert werden, wenn $|S_i| > 1$ und T_i mehr als ein Requirement realisiert, d. h. wenn mehr als ein realizedBy-Link bei T_i endet.

Es hat sich in den von uns bearbeiteten Fallstudien (siehe auch Abschnitt 8.2.2) allerdings herausgestellt, dass Interaktionen zwischen Needs, welche direkt durch den Task am Interaktionspunkt realisiert werden, nicht kritisch sind. Ein Beispiel dafür wäre obige Interaktion der Needs N1 und N2 in T1. Normalerweise entstehen

diese Interaktionen entweder durch zu feingranular spezifizierte Anforderungen oder durch orthogonale Anforderungen (wie N2, welcher zusätzlich eine Energieeinsparung fordert). Daher schließen wir an unseren Detektionsalgorithmus einen vierten Schritt an, in welchem geprüft wird, ob eine solche Situation vorliegt:

4. Ein Task T_i wird als Interaktionspunkt ausgeschlossen, wenn für alle Requirements R_l, die von Task T_i realisiert werden, gilt $R_l \in \{N_1, ..., N_m\}$.

Diese mathematische Formulierung des Algorithmus lässt sich sehr leicht mit Hilfe unserer Aktionssprache in ein Werkzeug umsetzen (wir wollen es *Analyzer* nennen), wenn wir die Tatsache ausnutzen, dass sowohl das Artefakt Task als auch das Artefakt Need eine Spezialisierung des Artefakts Requirement sind. Die Traversierung des Abhängigkeitsgraphen erfolgt daher im abstrakten Artefakt Requirement wie folgt:

```
1   /* Requirement: */
2   public class Requirement {
3       private Set S = new HashSet();
4       public Set getS() {
5           return S;
6       }
7       void traverseAndUpdateS(Need N) {
8           S.add(N);
9           foreach(Task T in realizingTask)
10              T.traverseAndUpdateS(N);
11      }
12  }
```

Zunächst erfolgt die Definition der Menge `S` (inkl. Zugriffsmethode) und deren Initialisierung (Schritt 1 von oben). Die Operation `traverseAndUpdate()` wird rekursiv definiert, s.d. alle von der aktuellen Requirement-Instanz realisierten Tasks erfasst werden und die jeweilige Menge `S` aktualisiert wird (Schritt 2 von oben).

Nun muss die `traverseAndUpdate()`-Methode nur noch für alle Needs der Spezifikation aufgerufen werden. Dies erfolgt in der Operation `detectInteractions()` im komplexen Artefakttyp RequirementsSpecification (vgl. Abschnitt 6.2.4 auf Seite 165):

```
1   /* RequirementsSpecification: */
2   public void detectInteractions() {
3       foreach(Need N in itsNeeds.itsNeed)
4           N.traverseAndUpdateS(N);
5
6       foreach(Task T in itsTaskList.itsTask) {
7           if( (T.getS().size() > 1) &&
8               (T.realizedRequirement.size() > 1) &&
```

```
9                !(T.realizesOnlyNeeds(itsNeeds.itsNeed)) ) {
10                   System.out.println("Interaktion von "+
11                    T.getS()+" in "+T.name);
12           }
13       }
14   }
```

Die Operation `realizesOnlyNeeds()` wird für die Realisierung des Schrittes 4 des mathematischen Algorithmus nötig und wird wie folgt definiert:

```
1   /* Task: */
2   public boolean realizesOnlyNeeds(Set allNeeds) {
3       Set realizedRequirements = this.realizedRequirement;
4       realizedRequirements.removeAll(allNeeds);
5       if(realizedRequirements.size() > 0)
6           return false;
7       else
8           return true;
9   }
```

Mit Hilfe der Java-Methode `removeAll()` berechnet man die asymmetrische Mengendifferenz der Menge aller von dem aktuellen Task realisierten Anforderungen (`realizedRequirements`) und der Menge aller Needs (`allNeeds`). Ist das Ergebnis eine nicht-leere Menge, so muss die Menge aller Anforderungen mindestens einen Task beinhalten und somit liefert die Operation `realizesOnlyNeeds()` als Ergebnis den Wert `false`.

Der Aufruf der obigen Methoden liefert dann im Falle unseres *RoomAutomation*-Beispiels die Ausgabe:

```
Interaktion von [N1, N2, N3] in T3
Interaktion von [N1, N2, N3] in T4
Interaktion von [N1, N2, N3] in T5
```

Wichtig ist anzumerken, dass auf Basis von Needs und Tasks alle *potenziellen* Feature-Interaktionen aufgedeckt werden können. Allerdings ist es möglich, dass nicht alle diese Interaktionen im endgültigen Produkt auftauchen. So könnte es sein, dass zur Laufzeit ein Task an einem Interaktionspunkt nie mehr als einen Need gleichzeitig realisiert.

Detektion auf Ebene der Objektstruktur

Um die Liste der potenziellen Feature-Interaktionen zu verfeinern, sollten Interaktionen, welche nicht auftreten können, aus der Liste der Interaktionen eliminiert werden. Ein erster Schritt in diese Richtung kann durch die Berücksichtigung der

Objektstruktur begangen werden. Da die Objektstruktur Informationen über die Instanziierung der Control-Object-Types beinhaltet, kann die Kenntnis über den „Einsatzort“ für eine solche Elimination eingesetzt werden. Eine Feature-Interaktion kann insbesondere immer dann ausgeschlossen werden, wenn abhängige Tasks nie an derselben Instanziierungsstelle verwendet werden. Abb. 8-3 illustriert eine solche Situation.

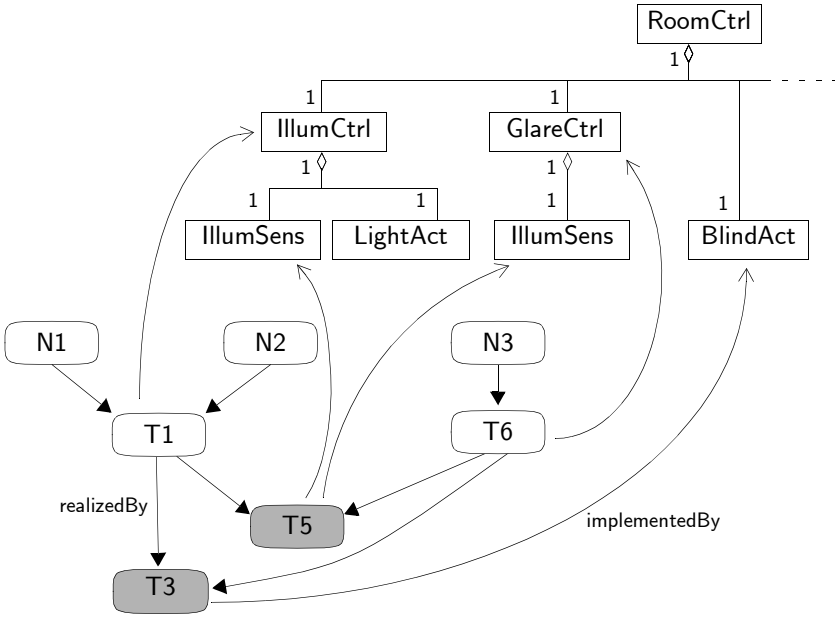


Abbildung 8-3. Elimination von Interaktionen auf Basis der Objektstruktur

Wie bereits in den Entwicklungsdokumenten des *RoomAutomation*-Beispiels festgehalten wurde, wird T5 von Control-Object-Type IllumSens implementiert (implementedBy-Relation in der Abbildung). Die abhängigen Tasks T1 und T6 aber werden von verschiedenen Objekttypen (IllumCtrl bzw. GlareCtrl) implementiert. Weil nun diese beiden Objekttypen jeweils eine individuelle Instanz des IllumSens-Objekttyps aggregieren und daher nur diese Instanz für die Bestimmung der Helligkeit heranziehen, kann Task T5 als Interaktionspunkt ausgeschlossen werden, da keine Interaktion zwischen T1 und T6 stattfinden kann.

Um solche Situationen algorithmisch feststellen zu können, können wir uns auf das Einhalten des Constraints *CS*3 aus Abschnitt 8.1.2 auf Seite 192 zurückziehen. Dieser impliziert, dass Control-Object-Types, welche abhängige Tasks implementieren, möglichst nahe beieinander liegen, weil ansonsten eine Routing über alle Zwischenknoten erfolgen müsste. Für die folgenden Betrachtungen lassen wir zur Vereinfachung diese „Routing“-Tasks weg, da sich diese zwangsläufig aus der Objektstruktur ergeben.

Zur Festzustellung, ob eine Interaktion auf dieser Ebene nun ausgeschlossen werden kann, betrachten wir uns die jeweils von einem Interaktionspunkt direkt reali-

sierten Tasks, da diese wegen obiger Annahme von den am nähesten instanziierten Objekttypen implementiert werden. Für die dann beteiligten Objekttypen wird deren Instanziierungsbeziehung berücksichtigt.

Sei zunächst $D(T_i)$ die Menge aller Tasks, die direkt von Task T_i realisiert werden und $P(T_i, T_j)$ die Menge aller Control-Object-Types, die auf den kürzesten Wegen zwischen dem Objekttyp, der T_i implementiert ($C(T_i)$), und dem Objekttyp, der T_j implementiert ($C(T_j)$), liegen. Existiert mehr als ein Pfad mit gleicher Länge, so werden die Control-Object-Types aller dieser Pfade in die Menge $P(T_i, T_j)$ aufgenommen. Die Bestimmung der zu eliminierenden Interaktionspunkte verläuft damit dann wie folgt:

1. Für jeweils alle Kombinationen zweier Tasks T_{j_1} und T_{j_2} mit $j_1 \neq j_2$ aus der Menge $D(T_i)$ berechnet man die Schnittmenge $I_{i,k} = P(T_i, T_{j_1}) \cap P(T_i, T_{j_2})$. Auf die Berechnung der Schnittmenge wird für diejenigen Pfade P verzichtet, welche nur einen Objekttyp beinhalten, da dann die abhängigen Tasks von demselben Control-Object-Type implementiert werden.
2. Einen Task T_i kann man als Interaktionspunkt ausschließen, wenn für alle k gilt $I_{i,k} = C(T_i)$. Dann nämlich sind keine gemeinsamen Objekttypen auf dem Pfad zwischen T_i und den realisierten Tasks zu finden und daher sind jeweils unterschiedliche Instanzen beteiligt.

Diesen Algorithmus wollen wir für das obige Beispiel von Task T5 veranschaulichen. Hier gilt $D(\mathsf{T5}) = \{\mathsf{T1}, \mathsf{T6}\}$ und damit

$P(\mathsf{T5}, \mathsf{T1}) = \{\mathsf{IllumSens}, \mathsf{IllumCtrl}\}$

und

$P(\mathsf{T5}, \mathsf{T6}) = \{\mathsf{IllumSens}, \mathsf{GlareCtrl}\}$,

was zu der Schnittmenge {IllumSens} führt. Daher kann T5 kein echter Interaktionspunkt sein.

Anders sieht es im Fall von Task T3 aus, für welchen die folgenden Pfade berechnet werden:

$P(\mathsf{T3}, \mathsf{T1}) = \{\mathsf{BlindAct}, \mathsf{RoomCtrl}, \mathsf{IllumCtrl}\}$

und

$P(\mathsf{T3}, \mathsf{T6}) = \{\mathsf{BlindAct}, \mathsf{RoomCtrl}, \mathsf{GlareCtrl}\}$,

was zu einer Schnittmenge führt, die mehr als $C(\mathsf{T3}) = \mathsf{BlindAct}$ beinhaltet. Somit kann T3 nicht als Interaktionspunkt ausgeschlossen werden. Dasselbe gilt für Task T4 (nicht in Abb. 8-3 gezeigt).

Die Umsetzung des obigen mathematischen Algorithmus in ein Werkzeug auf Basis unserer Aktionssprache AL++ ist weitgehend analog zu dem Vorgehen aus dem letzten Abschnitt. Insbesondere bzgl. der Schnittmengenberechnungen und dem Iterieren über alle relevanten Artefakte gleicht die Lösung der bereits vorgestellten Implementierung. Spannend ist allerdings die Berechnung der Menge $P(T_i, T_j)$, da

hierbei eine geschachtelte Rekursion notwendig wird und damit die Mächtigkeit einer operationalen Beschreibung nochmals deutlich zum Vorschein tritt (vgl. Abschnitt 4.2.1 auf Seite 62).

Zunächst die Implementierung der `determineShortestPath()`-Operation, welche P für zwei gegebene Tasks berechnet:

```
1   public Set determineShortestPath(Task Ti, Task Tj) {
2       ControlObjectType CTi = Ti.implementingControlObjectType;
3       ControlObjectType CTj = Tj.implementingControlObjectType;
4       SortedMap allPaths = new TreeMap();
5       determinePathsDownwards(CTi, CTj, new HashSet(), allPaths);
6       determinePathsUpwards(CTi, CTj, new HashSet(), allPaths);
7       Set shortestPath = new HashSet();
8       try {
9           Integer firstKey = (Integer)allPaths.firstKey();
10          shortestPath = (Set)allPaths.get(firstKey);
11      } catch(NoSuchElementException exc) {}
12      return shortestPath;
13  }
```

Die Berechnung des kürzesten Pfades beginnt bei $C(T_i)$, von wo aus alle ControlObjectTypes, die entlang der Aggregationshierarchie bis zum Objekttyp $C(T_j)$ zu finden sind, festgestellt werden. Die implementierenden Objekttypen lassen sich dabei leicht über die implements-Relation feststellen.

Um aus allen möglichen Pfaden am Ende den kürzesten bestimmen zu können, wird eine sortierte Tabelle (`SortedMap`) definiert, welche als Einträge die Pfade einer gewissen Länge und als Schlüssel (`key`) die entsprechende Größe der Menge der Objekttypen besitzt. Damit kann durch einen einfachen Zugriff das kürzeste „Element“ ausgewählt werden (Zeile 8–13). Falls kein Pfad gefunden wurde (weil evtl. nicht alle nötigen Entwicklungsinformationen vorhanden waren), wird die leere Menge zurückgeliefert. In Schritt 2 des obigen Algorithmus impliziert dies, dass auf Grund dieser fehlenden Information der entsprechende Interaktionspunkt zunächst nicht ausgeschlossen werden kann.

Die eigentliche Bestimmung der Pfade erfolgt über die Operationen `determinePathDownwards()` und `determinePathUpwards()`, welchen neben Start- und Zielobjekttyp und der Tabelle aller Mengen auch eine leere Menge als Initialwert übergeben wird. Dabei sucht die erste Operation zunächst nach unten im Aggregationsbaum, die zweite Operation richtet ihre Suche nach oben, wobei in jedem Elternknoten auch zusätzlich nach unten gesucht wird. Dies entspricht exakt den möglichen Wegen einer Signalweiterleitung in der PROBAnD-Methode: entweder befindet sich der Kommunikationspartner im aggregierten Teilbaum oder aber

die Kommunikation muss über den Elternknoten erfolgen, der das Signal an sein entsprechendes Ziel weiterleitet.

```
1   public void determinePathsDownwards(ControlObjectType currentCot,
2       ControlObjectType CTj, Set currentPath, SortedMap allPaths) {
3       if(currentCot == CTj) {
4           Integer size = new Integer(currentPath.size());
5           if(allPaths.containsKey(size)) {
6               Set s = (Set)allPaths.get(size);
7               s.addAll(currentPath);
8           } else {
9               allPaths.put(size, currentPath);
10          }
11          return;
12      } else {
13          ControlObjectType aggregatedCot;
14          Set newCurrentPath;
15          foreach(Instantiation inst in
16           currentCot.aggregatedInstantiation) {
17              aggregatedCot = inst.instantiatedControlObjectType;
18              newCurrentPath = new HashSet(currentPath);
19              newCurrentPath.add(aggregatedCot);
20              determinePathsDownwards(aggregatedCot, CTj,
21               newCurrentPath, allPaths);
22          }
23      }
24  }
```

Wird bei der Suche nach unten im Baum der gesuchte Control-Object-Type gefunden, so wird zunächst die Länge des Pfades bestimmt. Mit dieser Länge als Schlüssel lässt sich in der Tabelle `allPaths` feststellen, ob bereits ein anderer Pfad mit derselben Länge identifiziert wurde. Ist dies der Fall, werden die Objekttypen zu dieser bereits identifizierten Menge hinzugefügt. Anderenfalls wird der aktuelle Pfad als neuer Eintrag in die Tabelle aufgenommen.

Ist der gesuchte Objekttyp noch nicht gefunden, dann wird die Suche auf alle aggregierten Objekttypen (über die aggregates-Assoziation von ControlObjectType zu Instantiation) ausgedehnt. Dabei wird die Menge, welche den aktuellen Pfad beschreibt, jeweils dupliziert, da sich von hier ab die jeweiligen Objekttypen im Pfad unterscheiden.

Die Rekursion bricht entweder ab, wenn der gesuchte Objekttyp gefunden wurde oder aber alle erreichbaren Blattknoten besucht wurden. Im letzteren Fall greift dann die Operation `determinePathsUpwards()`, die nach der Operation `determine-`

`PathsDownwards()` in `determineShortestPath()` aufgerufen wird und die Suche nach oben hin ausdehnt:

```
1   public void determinePathsUpwards(ControlObjectType currentCot,
2    ControlObjectType CTj, Set currentPath, SortedMap allPaths) {
3       if(currentCot == CTj) {
4           /* siehe determinePathsDownwards() */
5       } else {
6           ControlObjectType aggregatingCot;
7           Set newCurrentPath;
8           foreach(Instantiation inst in
9            currentCot.instantiatingInstantiation)
10              aggregatingCot = inst.aggregatingControlObjectType;
11
12              newCurrentPath = new HashSet(currentPath);
13              newCurrentPath.add(aggregatingCot);
14              determinePathsDownwards(aggregatingCot, CTj,
15               newCurrentPath, allPaths);
16
17              newCurrentPath = new HashSet(currentPath);
18              newCurrentPath.add(aggregatingCot);
19              determinePathsUpwards(aggregatingCot, CTj,
20               newCurrentPath, allPaths);
21          }
22      }
23  }
```

Diese Operation ist sehr ähnlich zu derjenigen für die Suche nach unten. Unterschiede sind, dass die aggregates- und instantiates-Links in anderer Richtung verfolgt werden und dass für jeden Knoten eine Suche in beide Richtungen erfolgt. Die Rekursion endet hier daher zusätzlich falls kein weiterer Elternknoten gefunden wird.

Abb. 8-4 zeigt den Ablauf des Algorithmus an dem Beispiel zur Berechnung von $P(\mathsf{T5}, \mathsf{T1}) = \{\mathsf{IllumSens}, \mathsf{IllumCtrl}\}$.

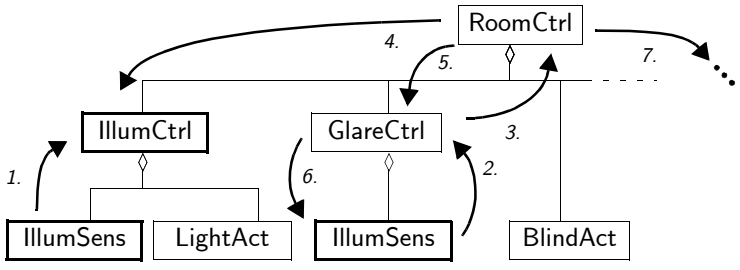


Abbildung 8-4. Berechnung des kürzesten Pfades am *RoomAutomation*-Beispiel

Der kürzeste Weg wird bereits im ersten Schritt gefunden, wenn der Link von IllumSens zu dem aggregierenden IllumCtrl verfolgt wird. Die anschließenden Schritte 2 bis 4 gehen den Weg über GlareCtrl und schließlich über RoomCtrl, wo sich die Suchrichtung umdreht. Schritte 5, 6 und 7 liefern keinen zusätzlichen Pfad, da der gesuchte Control-Object-Type nicht gefunden wird.

Detektion auf Strategieebene

Sobald in der Entwicklung begonnen wurde eine Realisierung der Tasks zu beschreiben, kann die Detektion von Feature-Interaktionen weiter verfeinert werden. Dazu betrachtet man die Kopplung zwischen Tasks, die durch die Definition von Strategien entstehen, welche über Signale oder Attribute Informationen austauschen.

Zwei Beobachtungen lassen sich anstellen, wenn man solche Informationen verwendet. Zunächst ist trotz der Detektion einer potenziellen Interaktion auf Anforderungsebene eine solche Interaktion nicht möglich, wenn bei der Kommunikation zwischen zwei Tasks einer der beiden immer nur Signale produziert, während der andere Task diese Signale nur konsumiert, also eine Kommunikation in ausschließlich einer Richtung erfolgt. Analoge Beobachtungen gelten für Attribute. Ein Beispiel für einen solchen Fall ist Task T4. Da der Bewegungssensor (MotionSens) nur Signale zur Benachrichtigung über eine gemessene Bewegung produziert und kein Signal empfangen kann, kann T4 kein Interaktionspunkt sein.

Die Detektion solcher Situationen ist sehr einfach, wenn man die Verfolgbarkeitsrelationen von den Strategien zu den Signaltypen bzw. Attributen ausnutzt. Unsere Erfahrung hat allerdings gezeigt, dass dadurch leider auch echte Interaktionen eliminiert werden, wenn die Entwicklungsdokumente nicht vollständig sind. So könnte es sein, dass noch nicht spezifiziert wurde, dass ein anderer Tasks ein Signal an den Bewegungsmelder sendet. Aus diesem Grund wurde die Implementierung der obigen Heuristik wieder aus unseren Werkzeugen entfernt.

Eine zweite Beobachtung bei der Berücksichtigung von Strategien ist, dass zusätzliche Interaktionen auftreten können. So kann ein Entwickler z. B. spezifizieren, dass die Strategien zweier Tasks, die nicht über eine realizedBy-Relation verbunden sind, auf derselben Menge von Daten arbeiten, indem beide gewisse Attribute lesen bzw. schreiben. Dies führt zu einer Interaktion, die zuvor nicht detektiert wurde.

Tatsächlich deutet die Detektion solcher Interaktionen auf unvollständige Anforderungsdokumente hin, da die Abhängigkeit, welche durch eine solche Kopplung eingeführt wird, in Wirklichkeit eine realizedBy-Beziehung widerspiegelt. Als Reaktion auf diese Feststellung wurde der Detektionsalgorithmus insofern erweitert, als dass eine durch Attribute eingeführte Kopplung als „hypothetischer“ realizedBy-Link „sichtbar“ gemacht wird. Danach kann die Detektion solcher Interaktionen durch Aufruf der `detectInteractions()`-Operation der Anforderungsebene durchgeführt werden.

Detektion auf Umgebungsebene

Wie bereits in der Einführung zu diesem Abschnitt dargelegt wurde, spielt die Umgebung für Feature-Interaktionen in MSR-Systemen eine bedeutende Rolle, da die physikalische Umgebung die Quelle für implizite Kopplungen verschiedener Teile des Systems sein kann.

Bereits in unserem kleinen *RoomAutomation*-Beispiel lässt sich dies nachweisen. Der Abhängigkeitsgraph aus Abb. 8-2 auf Seite 210 zeigt keinerlei Verbindungen zwischen dem Teil der Lichtsteuerung (Needs N1 bis N3) und dem Teil der Temperaturregelung (Need N4). Allerdings besteht eine physikalische Abhängigkeit zwischen der Raumtemperatur und dem Anteil des Sonnenlichts, welches in den Raum gelangt. Wie in der Einleitung bereits illustriert wurde, führt die Sonnenstrahlung zu einer Erwärmung des Raumes. Daher wird in der Realität eine Feature-Interaktion zwischen N4 und den anderen Needs des *RoomAutomation*-Systems beobachtet.

Solche Arten der Interaktion müssen also zwingend bei der Entwicklung von MSR-Systemen berücksichtigt werden. Um die Detektion auf dieser Ebene automatisieren zu können, müssen die physikalischen Wechselwirkungen der Umgebung explizit gemacht werden.

In dieser Situation kommt es uns zu Gute, dass wir in Abschnitt 6.1.1 auf Seite 132 bereits die Existenz von Umgebungssimulatoren für einen sinnvollen Test von reaktiven Systemen gefordert hatten. Da solche Simulatoren die physikalischen Wechselwirkungen berücksichtigen, um realistische Simulationsergebnisse zu liefern, beinhalten die Simulatorspezifikationen (-modelle) die für eine Feature-Interaktionsdetektion notwendigen Abhängigkeiten. In unserem Fall können wir dabei sogar auf Simulatoren zurückgreifen, welche in dem Formalismus der PROBAnD-Methode spezifiziert wurden, was diese Aufgabe weiter vereinfacht. Dazu gehören Gebäudesimulatoren (siehe [Zim02] und [MMZ02]) aber auch ein „Vehicle-Dynamics"-Simulator (vgl. [Cal03]).

Die Schnittstelle eines reaktiven Systems zu dessen Umgebung wird durch Sensoren und Aktuatoren realisiert (siehe Abschnitt 6.1.1 auf Seite 132). Daher muss für jede solche Komponente ein entsprechendes Gegenstück im Simulator existieren, das die Funktionalität des physikalischen Sensors bzw. Aktuators nachahmt. Folglicherweise, wird jeder Task eines reaktiven Systems, der durch einen Sensor- oder Aktuator-Control-Object-Type implementiert wird, schlussendlich durch einen oder mehrere Tasks des zugehörigen Simulator-Objekttyps realisiert. Da die realizedBy-Relationen zwischen den Simulator-Tasks die physikalischen Wechselwirkungen widerspiegeln, führt eine Interaktionsdetektion mit Hilfe dieser Tasks zu einer Liste von Feature-Interaktionen, welche durch die Umgebung bedingt sind.

Die oben skizzierte Situation ist nochmals in Abb. 8-5 am Beispiel des *RoomAutomation*-Systems illustriert.

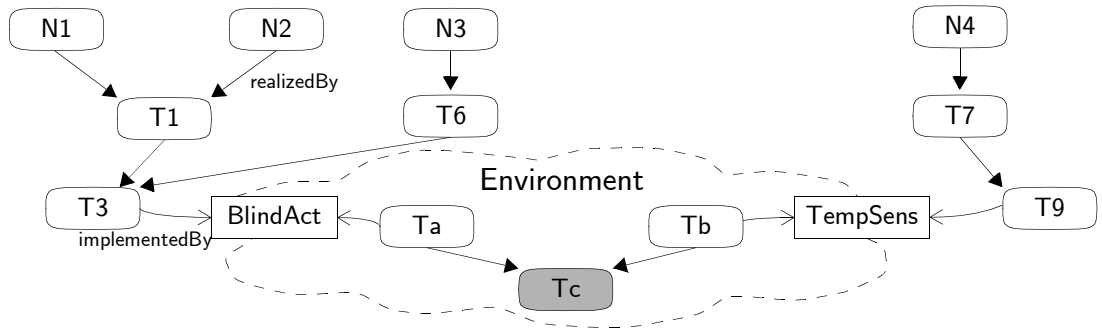


Abbildung 8-5. Physikalische Kopplung von Licht- und Temperaturregelung

Wir wissen bereits, dass T3 von BlindAct und T9 von TempSens implementiert wird. Nehmen wir nun an, dass der Task für die Simulation des Jalousie-Aktuators Ta sei und der Task zur Simulation des Temperatursensors Tb. Nehmen wir nun weiter an, dass beide Tasks durch Tc realisiert werden, welcher die Raumtemperatur ausgehend von der Jalousiestellung berechnet, dann kann dieser Task als Interaktionspunkt identifiziert werden. Als Folge wird eine Interaktion zwischen N1, N2, N3 und N4 durch die physikalische Umgebung festgestellt.

Um eine solche Detektion vollautomatisch realisieren zu können, muss eine systematische Verschmelzung der Spezifikation des MSR-Systems und der Spezifikation der Umgebung erfolgen. Während einer solchen Verschmelzung werden zunächst die Sensoren und Aktuatoren als Verbindungspunkte ausgemacht. Zur Vereinfachung der Identifikation übereinstimmender Sensoren und Aktuatoren im reaktiven System und der Umgebung gehen wir in unserer Arbeit davon aus, dass die Komponenten in beiden Systeme dieselben Namen besitzen. Ansonsten wäre eine entsprechende Umbenennung in einem der beiden Systeme einfach möglich.

Für jedes Paar von implementedBy-Links von einem Control-Task zu einem Control-Object-Type und von einem Simulator-Objekttyp zu einem Simulator-Task wird wieder ein „hypothetischer“ realizedBy-Link eingeführt, welcher die beiden Teilsysteme verbindet. Nach diesem Schritt kann dann wieder der initiale `detectInteractions()`-Algorithmus für die Interaktionsdetektion herangezogen werden.

In der Sprache AL++ gestaltet sich die Verschmelzung der beiden Systeme durch die Operation `mergeSystemAndEnv()` des komplexen Artefakttyps RequirementsSpecification wie folgt:

```
1   /* RequirementsSpecification: */
2   public void mergeSystemAndEnv(RequirementsSpecification rsEnv) {
3       ControlObjectType cotSystem;
4       ObjectStructure osEnv = rsEnv.itsObjectStructure;
5       foreach(ControlObjectType cotEnv in osEnv.itsControlObjectType) {
```

```
6           cotSystem = null;
7           foreach(cotSystem in itsObjectStructure.itsControlObjectType) {
8               if(cotSystem.name.equalsIgnoreCase(cotEnv.name))
9                   break;
10          }
11          if((cotSystem != null) &&
12             (cotSystem.aggregatedInstantiation.size() == 0)) {
13              foreach(Task taskSystem in cotSystem.implementedTask) {
14                  foreach(Task taskEnv in cotEnv.implementedTask) {
15                      taskEnv.realizedRequirement += taskSystem;
16                      taskEnv.name = taskEnv.name+"_ENV";
17                      itsTaskList.itsTask += taskEnv;
18               }
19          }
20      }
21  }
```

Als Übergabeparameter erhält diese Operation einen Verweis auf die Requirements-Spezifikation der Umgebung (`rsEnv`), deren Tasks es mit der aktuellen RequirementsSpecification zu verschmelzen gilt.

Zunächst werden dazu Control-Object-Types in beiden System gesucht, welche einen identischen Namen besitzen (Zeilen 3 bis 10). Wurde eine solche Übereinstimmung gefunden, dann wird geprüft ob dieser Objekttyp als Sensor bzw. Aktuator in Frage kommt. Dies ist in der momentanen Version des Produktmodells nur mit der Annahme möglich, dass ein Blattknoten entweder ein Sensor oder ein Aktuator ist. Alternativ könnte man natürlich auch eine Liste der Namen der echten Sensor- und Aktuator-Objekttypen angeben.

Als letzer Schritt wird dann für jeden Task eines Control-Object-Types (`cotSystem`) ein realizedBy-Link zu allen Tasks des korrespondierenden Simulator-Objekttyps (`cotEnv`) angelegt. Um bei der späteren Ausgabe der Interaktionspunkte die Tasks des reaktiven Systems von denen der Umgebung unterscheiden zu können hängen wir hier der Einfachheit halber noch den String „`_ENV`“ an den Namen des Tasks an. Der Eintrag des Tasks `taskEnv` in die Taskliste des reaktiven Systems ist notwendig, da für Detektion der Interaktionspunkte alle Tasks bekannt sein müssen (siehe Abschnitt zur Detektion auf Anforderungsebene).

Betrachtet man sich die typischen Abhängigkeitsgraphen der Needs und Tasks eines Simulators, so wird man allerdings feststellen, dass dieser Graph eine ähnliche Struktur wie der Graph eines Steuer- oder Regelsystems aufweist, d.h. der Graph „verbreitert“ sich ausgehend von den Needs nach unten. In Abb. 8-5 auf Seite 220 hatten wir eine andere Struktur illustriert, was uns zunächst die Erläuterung des Grundkonzepts vereinfachte. Um nun realistische Simulatormodelle verwenden zu

können, muss dem eigentlichen Verschmelzungsschritt ein Schritt vorausgehen, der die realizedBy-Links zwischen den Tasks des Simulator umkehrt (damit erreichen wir dann eine solche Graphstruktur wie in Abb. 8-5).

Eine solche Umkehrung lässt sich durch folgende Operation des komplexen Artefakttyps RequirementsSpecification realisieren:

```
1   /* RequirementsSpecification: */
2   public void redirectTasks() {
3       ObjectStructure osNew = new ObjectStructure("ObjectStructure");
4       TaskList tlNew = new TaskList("TaskList");
5       Hashtable taskTable = new Hashtable();
6       Hashtable cotTable = new Hashtable();
7       ControlObjectType newCot;
8       ControlObjectType oldCot;
9       foreach(Task oldTask in itsTaskList.itsTask) {
10          newTask = retrieveNewReference(taskTable, oldTask.name,
11           "Task", tlNew);
12          oldCot = oldTask.implementingControlObjectType;
13          newCot = retrieveNewReference(cotTable, oldCot.name,
14           "ControlObjectType", osNew);
15          Task newRealizedTask;
16          foreach(Requirement req in oldTask.realizedRequirement) {
17              if(req.type == Task) {
18                  Task oldRealizedTask = (Task)req;
19                  newRealizedTask = retrieveNewReference(taskTable,
20                   oldRealizedTask.name, "Task", tlNew);
21                  newTask.realizingTask += newRealizedTask;
22              }
23          }
24          newCot.implementedTask += newTask;
25      }
26      itsObjectStructure = osNew;
27      itsTaskList = tlNew;
28  }
```

Die Änderung existierender realizedBy-Beziehungen ist relativ aufwändig, da man sehr genau verwalten muss, welche der Relationen man bereits umgekehrt hat. Daher wählen wir hier die Lösung, die nötigen Artefakte neu zu erzeugen. Da nur die Control-Object-Types und die Tasks für unseren obigen Algorithmus relevant sind, gestaltete sich das als recht einfach.

Es wird dazu systematisch für jede Instanz des Artefakttyps ControlObjectType eine neue Instanz mit identischem Namen angelegt und diese zur neu erzeugten Instanz des komplexen Artefakttyps ObjectStructure hinzugefügt. Analog verfährt

man für die von einem Objekttyp implementierten Tasks. Eine neue Taskliste ist nötig, weil wir diese neuen Tasks bei der Verschmelzung in die Taskliste des MSR-Systems aufnehmen müssen.

Die eigentliche Umkehrung erfolgt in Zeile 21, wo ein ursprünglich von einem Task *realisierte* Task (realizedTask) als ein von diesem Task *realisierter* Task (realizingTask) beschrieben wird.

Die Operation `retrieveNewReference()` erzeugt bei Bedarf eine neue Instanz oder liefert eine bereits erzeugte Instanz mit dem gewünschten Namen (und Typ) zurück. Sie ist wie folgt definiert:

```
1   /* RequirementsSpecification: */
2   private ArtefactType.anyinstance retrieveNewReference(Hashtable table,
3    String name, String typeName, ComplexArtefactType.anyinstance ca) {
4       ArtefactType.anyinstance temp;
5       if(!table.containsKey(name)) {
6           temp = new #typeName(name);
7           table.put(name, temp);
8           String roleName = "its"+typeName;
9           ca.#roleName += temp;
10      } else {
11          temp = table.get(name);
12      }
13      return temp;
14  }
```

Zunächst wird eine temporäre Variable `temp` definiert, welche Referenzen auf beliebige Instanzen eines Artefakts aufnehmen kann. Befindet sich eine Instanz mit dem gewünschten Namen nicht in der angegebenen Tabelle (`table`), dann wird eine Instanz des entsprechenden Typs (`typeName`) erzeugt und in die Tabelle eingetragen. Ebenso wird dieses Artefakt von dem angegebenne komplexen Artefakt `ca` aggregiert. Wir machen hier von dem generischen „#“-Operator von AL++ Gebrauch. Existiert die gewünschte Instanz bereits, so wird deren Referenz zurückgeliefert.

Die gesamte Interaktionsdetektion auf Umgebungsebene lässt sich dann durch die folgende in der RequirementsSpecification definierten Operation beschreiben:

```
1   /* RequirementsSpecification: */
2   public void detectInteractionsEnv(RequirementsSpecification rsEnv) {
3       rsEnv.redirectTasks();
4       mergeSystemAndEnv(rsEnv)
5       detectInteractions();
6   }
```

Als Ausgabe liefert diese Operation dann die folgende zusätzliche Zeile:

```
Interaktion von [N1, N2, N3, N4] in Tc_ENV
```

Interessanterweise deuten auch hier zusätzlich detektierte Interaktionen auf unvollständige Anforderungsdokumente hin. Denn ohne eine explizite Berücksichtigung der Interaktionen im Steuer- und Regelsystem besteht keine Möglichkeit korrekt auf die physikalischen Wechselwirkungen zu reagieren. Insbesondere bedeutet dies, dass Tasks existieren müssen, bei deren Spezifikation die physikalischen Wechselwirkungen berücksichtigt wurden. In unserem kleinen Raumbeispiel würde man z.B. einen Task erwarten, welcher die Verwendung von Tageslicht verbietet, wenn dies mit der Temperaturregelung in Konflikt steht.

8.2.2 Fallstudien

Obige Konzepte und die daraus entstandenen Werkzeuge wurden bzgl. ihrer Praktikabilität und Anwendbarkeit in verschiedenen Fallstudien untersucht. Dabei wurden verschiedene Domänen von reaktiven Systemen behandelt. Die betrachteten Systeme schließen zwei Gebäudeautomationssysteme, einen Automotive-Controller und eine Steuerung einer Eisenbahnkreuzung mit ein.

Eine ausführlich Diskussion dieser Fallstudien ist in [Met04] zu finden. Wir möchten in dieser Arbeit nur die Fallstudien zur Gebäudeautomation herausgreifen, da wir uns im weiteren Verlauf auf diese beziehen werden.

Fallstudien zur Gebäudeautomation

In der ersten betrachteten Fallstudie wurde eine Temperatur- und Lichtsteuerung, *FloorAutomation* genannt, für ein gesamtes Stockwerk eines Bürogebäudes betrachtet. Eine genaue Beschreibung des Systems findet sich in [QuZ99] und [Que02, S.341]. Auf die Analyse quantitativer Entwicklungsdaten werden wir in Abschnitt 10.2 auf Seite 309 bei der Betrachtung des Automatisierungsgewinns eingehen.

Um einen Eindruck des *FloorAutomation*-Systems zu gewinnen, ist in Tabelle 8-1 ein Auszug aus der Liste der 67 Needs gezeigt. Zur Realisierung der 67 Needs von *FloorAutomation* wurden insgesamt 233 Tasks und 37 Control-Object-Types spezifiziert.

Eine Erweiterung dieses Systems, genannt *FloorAutomationX*, wurde in der zweiten Fallstudie (siehe [QTB02]) bearbeitet und berücksichtigt neben Beleuchtung und Heizung auch ein Alarmsystem. Diese Tatsache spiegelt sich in 12 zusätzlichen Needs wider, von denen drei in Tabelle 8-1 aufgeführt sind. Desweiteren wur-

den 19 Tasks und drei Control-Object-Types zur Realisierung der neuen Features spezifiziert.

Tabelle 8-1. Needs zweier komplexer Gebäudeautomationssysteme

	Need	*Description*
Beleuchtung *FloorAutomation*, *FloorAutomationX*	U2	As long as the room is occupied the chosen light scene has to be maintained.
	FM1	Use daylight to achieve the desired illumination whenever possible.
	FM6	The facility manager can turn on/off any light in a room or hallway section.
Heizung *FloorAutomation*, *FloorAutomationX*	UH2	The comfort temperature shall be reached as fast as possible during heating up and shall be maintained as best as possible afterwards.
	FMH2	The use of solar radiation for heating should be preferred against using the central heating unit.
Alarm *FloorAutomationX*	UA2	If a person occupies a room with an activated alarm system, he can deactivate the alarm system by identifying himself within t_{alarm} seconds. Otherwise, the alarm must be triggered.
	UA10	If an alarm is triggered, all lights in the corresponding sections are turned on. When the alarm is reset, the lights are reset to their previous state.
	FA1	The facility manager can switch off an alarm, deactivate an alarm system, and activate the alarm system for an individual room or for all rooms of the building.

Die Zahlen der jeweils detektierten Feature-Interaktionen sind in Tabelle 8-2 dargestellt.

Tabelle 8-2. Ergebnisse der Feature-Interaktionsdetektion

Ebene	**Anzahl Feature-Interaktionen**			**Anzahl Interaktionspunkte**		
	Floor Automation	**Floor AutomationX**		**Floor Automation**	**Floor AutomationX**	
Anforderungen	18	21		25	27	
Objektstruktur	17	19		22	24	
Strategien	—	—	41	—	—	74
Umgebung	23	26	44	52	54	91

Eine Interaktionsdetektion auf Strategieebene konnte für *FloorAutomation* nicht durchgeführt werden, da in den entsprechenden Dokumenten keine Attribute beschrieben waren. Um die Ergebnisse mit denen von *FloorAutomationX* vergleichen zu können, wurde die Interaktionsdetektion für *FloorAutomationX* zusätzlich auch ohne Attribute durchgeführt (erste Spalte der *FloorAutomationX*-Subtabelle).

Typische Interaktionen, die in *FloorAutomation* festgestellt wurden, sind [U2, FM1, FM6] und [UH2, FMH2]. Diese Interaktionen sind relativ offensichtlich, da die interagierenden Needs verschiedene Aspekte eines gemeinsamen Features beschreiben. Dies liegt daran, dass viele Needs dieser Fallstudie sehr feingranular und lösungsorientiert formuliert wurden (die Systementwickler waren gleichzeitig auch die „Kunden").

Bei der Erweiterung des Systems zu *FloorAutomationX* wurden sieben neue Interaktionen eingeführt, die sich bereits auf der Anforderungsebene manifestieren. Eine dieser Interaktionen tritt zwischen UA2 und FA1 auf, was innerhalb des Alarmsystems liegt. Zusätzlich wurden Feature-Interaktionen auch zwischen den einzelnen Subdomänen entdeckt. So zwischen U2 und UA10. Diese Interaktion tritt auf, weil sowohl Need U2 als auch Need UA10 die Beleuchtung für deren Realisierung benötigen. Dies offenbart das Problem, dass sobald ein Alarm ausgelöst wird, die gewählte Lichtszene nicht länger – wie in U2 gefordert – aufrecht erhalten werden kann und stellt somit einen Konflikt mit den ursprünglichen Anforderungen dar.

Mit Kenntnis der Objektstruktur reduziert sich die Anzahl der detektierten Interaktionen um zwei (die Anzahl der Interaktionspunkte um drei). Ein Beispiel für eine eliminierte Interaktion ist [U2, UA2]. Diese Interaktion wurde auf der Anforderungsebene festgestellt, da sowohl die Lichtsteuerung als auch das Alarmsystem jeweils den Control-Object-Type Contact verwenden. Da die Lichtsteuerung allerdings eine Instanz dieses Contacts zum Feststellen, ob eine Leuchte eingeschaltet wird nutzt, und das Alarmsystem eine andere Instanz nutzt, um das unerlaubte Öffnen eines Fensters (Fensterkontakt) festzustellen, kann diese Interaktion auf Ebene der Objektstruktur eliminiert werden.

Zuletzt wurden bei Einbeziehen der Umgebung drei zusätzliche Interaktionen in *FloorAutomationX* aufgedeckt. Beispielsweise besteht eine solche Interaktion zwischen FM1 und UH2 weil eine physikalische Wechselwirkung zwischen dem Control-Object-Type TempSens, der zur Realisierung von Need UH2 beiträgt, und dem Objekttyp BlindAct, welcher den Lichtsteuerungs-Need FM1 realisiert, besteht.

8.2.3 Diskussion

Wie man auch an den Ergebnissen dieser Fallstudien erkennt, wird durch jede neue Ebene der Information die Kenntnis über Feature-Interaktionen verfeinert. Allerdings bedeutet dies weder, dass die Interaktionen einer Ebene die Interaktionen der vorhergehenden Ebene *aus*schließen (eine potenzielle Interaktion auf Anforderungsebene kann auch auf Umgebungsebene bestehen), noch dass die Interaktionen einer Ebene die Interaktionen der vorhergehenden Ebene *ein*schließen (eine Interaktion auf Anforderungsebene kann auf Ebene der Objektstruktur eliminiert werden).

Weiter ist es wichtig an dieser Stelle darauf hinzuweisen, dass obwohl eine Detektion von Feature-Interaktionen während der Entwicklung möglich und auch überaus sinnvoll ist, die Daten der Fallstudien erst nach Fertigstellen des Gesamtsystems berechnet wurden. Daher ist es anzunehmen, dass sich die Zahlen der obigen Feature-Interaktionen und auch die aufgedeckten unerwünschten Interaktionen bei einer prozessbegleitenden Anwendung des Detektionswerkzeuges anders darstellen würden.

Unerwünschte und notwendige Interaktionen

Einige der Interaktionen, die in den obigen Fallstudien aufgedeckt wurden, scheinen nicht kritisch zu sein, so z.B. [U2, FM1, FM6]. Man könnte meinen, auch solche Interaktionen sollten aus der Liste der Interaktionen eliminiert werden. Dies ist allerdings nur dann angebracht, wenn es sich bei der betrachteten Entwicklungsaktivität um eine echte Erweiterung des Systems handelt und daher *unerwünschte* Interaktionen detektiert werden müssen. Für andere Entwicklungsaktivitäten ist dies eventuell nicht sinnvoll.

Insbesondere bei einer Wiederverwendung von alten Systemteilen bei der Konstruktion eines neuen Systems ist eine detaillierte Kenntnis über die *notwendigen* Wechselwirkungen, die natürlich auch Feature-Interaktionen sind, unumgänglich. Ansonsten könnten relevante Teile des wiederverwendeten Systems vergessen werden, was zu einer Fehlfunktion der Wiederverwendungsartefakte in ihrem neuen Kontext führen kann. Als Beispiel für eine solche notwendige Interaktion wollen wir annehmen, dass unser kleines *RoomAutomation*-Beispiel um die in Abschnitt 8.2.1 angesprochene Konfliktlösungskomponente erweitert worden wäre. Damit führen wir die Feature-Interaktion [N1, N2, N3, N4] bereits auf Anforderungsebene ein. Bei der Erstellung eines neuen Systems, das nur der Temperaturregelung dienen soll, könnte man nun Teile unseres *RoomAutomation*-Systems verwenden. Bei oberflächlicher Betrachtung würde es ausreichen, einfach alle Komponenten, die für eine Lichtsteuerung relevant sind (insbesondere die Objekttypen LightAct und BlindAct), zu entfernen. Bei einer solchen Aktivität würde man dann aber die notwendige Interaktion zwischen der Temperaturregelung und der Jalousie übergehen und damit eine Temperaturregelung erhalten, die ein schlechtes Verhalten in einem sonnendurchfluteten Raum zeigt.

Leider kann die Unterscheidung zwischen notwendigen und unerwünschten Interaktionen auf der Basis der im Produktmodell verfügbaren Informationen nicht automatisch durchgeführt werden, sondern muss von den Entwicklern getroffen werden. Dennoch ist die Komplexitäts- und Aufwandsreduktion der Interaktionsdetektion durch die obigen Strategien enorm. So musste man in *FloorAutomationX* nur 21 Interaktionen (in 27 Interaktionspunkten) im Vergleich zu 331 Anforderungen, die über 298 realizedBy-Links verbunden sind, untersuchen.

Zur Illustration der Komplexität ist der Abhängigkeitsgraph für *FloorAutomationX* in Abb. 8-6 gezeigt.

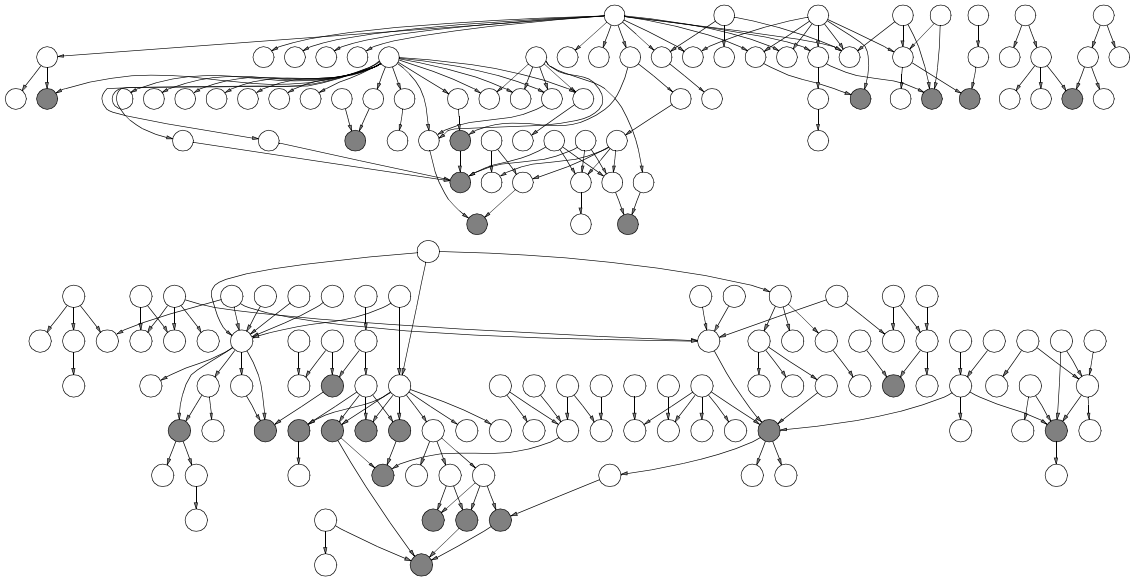


Abbildung 8-6. Abhängigkeitsgraph für *FloorAutomationX*

In diesem Graphen sind nur die von mindestens einem Task realisierten Needs aufgenommen. Jeder dieser Needs bildet die „Wurzel“ eines Teilgraphen (z.B. der Kreis in der rechten oberen Ecke der Abbildung). Die grau schattierten Kreise kennzeichnen die Interaktionspunkte.

Verwandte Arbeiten

Wilson et al. betrachten als einige wenige Autoren in [WiM03] Feature-Interaktionen in reaktiven Systemen, indem sie die Domäne der Heimautomationssysteme untersuchen. Die Wichtigkeit der Umgebung solcher Systeme wird erkannt und der Fokus der Ansätze daher auf die Sensoren und Aktuatoren (Geräte) gerichtet. Als Lösung des Interaktionsproblems wird eine Laufzeitlösung („on-line“) vorgeschlagen, die sog. „Device Manager“ zum Einsatz bringt, welche als „Feature Manager“ verstanden werden können (vgl. [CKM03]). Ein solcher „Device Manager“ weist alle Anfragen, die zu unerwünschten Interaktionen führen könnten, auf der Basis eines Umgebungsmodells ab. Ein Umgebungsmodell beinhaltet dabei die betrachteten physikalischen Größen und deren Einfluss (positiv oder negativ) auf das jeweilige Gerät. Durch Einsatz von Konzepten aus dem Bereich der Betriebssysteme (wie z.B. Monitore oder Semaphoren), können verschiedene Strategien zur Konfliktlösung implementiert werden.

Ein solcher Laufzeiteinsatz bietet eine sehr große Flexibilität, wenn neue Geräte während der Betriebszeit des Systems hinzugefügt werden. Falls aber ein neues Gerät bisher nicht betrachtete physikalischen Größen beeinflusst oder einen bisher un-

bekannten Einfluss auf schon betrachtete Größen zeigt, müssen die „Device Manager“ neu konstruiert werden (die Umgebungsmodelle müssen angepasst werden). Dies ist natürlich eine „off-line“ Aktivität. Desweiteren bietet der Ansatz von Wilson et al. keine genauere Behandlung von notwendigen Interaktionen, die bei dem Entfernen eines Gerätes eigentlich Berücksichtigung finden müssten.

Im Bereich der Telekommunikationssysteme wurden vielzählige Interaktionsdetektionsansätze in der Literatur vorgeschlagen. Insbesondere die Proceedings des oben bereits erwähnten „Feature Interaction Workshops“ (siehe u.a. [MaC00] und [AmL03]) und die Übersichtspapiere von Calder et al. [CKM03] und Keck et al. [KeK98] bieten sehr gute Verweise auf interessante Arbeiten in diesem Gebiet.

Die physikalische Umgebung spielt bei Telekommunikationssysteme eine untergeordnete Rolle, da ein Wechselwirkung immer nur zwischen Telekommunikationsgeräten (z.B. Mobiltelefon oder Fax-Gerät) stattfindet und keine physikalischen Größen relevant beeinflusst werden. Daher bezweifeln wir, ob man die obigen Lösungen aus der Telekommunikationsdomäne ohne größere Anpassungen für die Domäne der MSR-Systeme einsetzen kann.

Alle Beiträge aus dem Bereich der Telekommunikation lassen sich in zwei Kategorien einteilen: in solche, bei denen die Eingabe in den Detektionsprozess eine Systemspezifikation (abstraktes Modell) ist, und solche, bei denen die Eingabe aus Implementierungs-Code besteht. Die letzteren Ansätze besitzen den Nachteil, erst sehr spät in der Entwicklung eingesetzt werden zu können. Daher können sich unerwünschte Interaktionen schon in das System „eingeschlichen“ haben und deren Beseitigung entsprechend teuer werden. Desweiteren ist die Detektion nicht ganz einfach, da Code oftmals nicht semantisch „reich“ genug für eine sinnvolle Analyse ist.

In unseren Augen sollte man daher die erste Kategorie von Lösungen bevorzugen. Einen umfangreichen Vergleich aktueller Vorschläge aus dieser Kategorie findet man wieder in [CKM03]. Als Beispiel sei hier die Arbeit von Amyot et al. [ACG00] genannt, in welcher unerwünschte Feature-Interaktionen durch den Einsatz von *Use Case Maps* (*UCM*s, siehe [Buh98]) und LOTOS aufgedeckt werden. Bei den UCMs handelt es sich um eine in Standardisierung befindliche Notation zur Beschreibung von Szenarien (vgl. Abschnitt 9.1.4 auf Seite 254), mit welcher insbesondere Wechselwirkungen zwischen Szenarien beschrieben werden können. Nach einer teil-formalen Spezifikation von Features mit Use Case Maps werden diese im Ansatz von Amyot et al. mit Hilfe von *LOTOS* (*Language of Temporal Ordering Specifications*, siehe [BoB86]) formalisiert und dann einer Verifikation unterzogen.

Nicht nur in den Arbeiten von Amyot et al. sondern auch in vielen der anderen Lösungen muss ein Modellierer oder Entwickler explizit und teilweise formal die in einem System betrachteten Feature spezifizieren. Bei der von uns vorgeschlagenen Lösung ist dieser zusätzliche Schritt nicht nötig und daher kann man von einem

Qualitätsgewinn bereits auf der Basis der Dokumente des existierenden PROBAnD-Prozesses profitieren.

8.3 Automatisierte Bewertung von Artefakten

Wie wir an dem vorausgegangenen Problem der Feature-Interaktion sahen, ist eine Messung und Bewertung der Merkmale von Artefakten bereits *während* der Entwicklung sinnvoll, um eventuelle korrektive Maßnahmen einleiten zu können, wenn die Qualität der Produkte nicht den gewünschten Anforderungen entspricht. Eine sehr einfache Bewertung könnte z.B. durch die Überprüfung der Komplexität einzelner Control-Object-Types erfolgen, um in der PROBAnD-Methode rechtzeitig eine Aufspaltung eines zu komplexen Objekttyps in kleiner Komponenten vornehmen zu können. Besonders spannend ist eine automatische Bewertung von Artefakten, weil sie den Entwicklern eine sofortige und aufwandsneutrale Rückmeldung bezüglich der Produkteigenschaften liefern kann.

Beginnen wollen wir diesen Abschnitt mit der Berechnung eines komplexen Maßes, welches sich auf die im vorangegangenen Abschnitt berechnete Zahl von Feature-Interaktionen stützt. Neben solchen komplexen Maßen wird im Anschluss daran eine Messvorschrift vorgestellt, die sehr einfach zu implementieren ist. Für die in diesem Zusammenhang relevanten Grundlagen zur Messung sei auf Abschnitt 10.1 auf Seite 301 verwiesen.

8.3.1 Bewertungen an Hand der Feature-Interaktionen

Wie wir in Abschnitt 8.2.1 auf Seite 209 erwähnten, entschlossen wir uns bei der Entwicklung des Werkzeugs zur Interaktionsdetektion, diejenigen Needs, welche direkt von dem Task am Interaktionspunkt realisiert werden, von einer weiteren Betrachtung auszuschließen. Begründet wurde dies unter anderem damit, dass solche Needs zu feingranular sind. Damit bietet sich ein sehr interessantes Maß zur Beurteilung der Qualität der Anforderungsdokumente an.

Sei n_{FI} die Anzahl aller Feature-Interaktionen und $\hat{n}_{\mathrm{FI}}$ die Zahl der Interaktionen, die auf Grund der „verbesserten“ Betrachtung ermittelt wurden. Dann lässt sich ein Quotient φ berechnen zu:

$$\varphi = \frac{n_{\mathrm{FI}} - \hat{n}_{\mathrm{FI}}}{n_{\mathrm{FI}}}, \qquad \textbf{(Gleichung 8-1)}$$

Dabei deutet ein Wert von φ, der wesentlich von 0 abweicht, darauf hin, dass die Anforderungen möglicherweise zu feingranular formuliert wurden, was später zu eventuellen Problemen bei der Verfolg- und Änderbarkeit des Systems führen kann. Für die Fallstudie *FloorAutomationX* liegt diese Zahl bei 0,49 ($n_{\mathrm{FI}} = 41$, siehe

[MeW03] und $\hat{n}_{\mathrm{FI}} = 21$, siehe Tabelle 8-2 auf Seite 225), was die dort erwähnte feingranulare Spezifikation der Needs unterstreicht.

Ein weiteres interessantes Maß wäre die Anzahl der Interaktionspunkte in Relation z.B. zu der Komplexität oder Größe (z.B. Anzahl Needs oder Tasks) des Gesamtsystems zu betrachten, da die relative Anzahl der identifizierten Interaktionspunkte Rückschlüsse auf die Qualität der Realisierung zulassen kann. Sehr wichtig bei diesem Maß ist aber zunächst ausreichendes Datenmaterial zu sammeln, um beurteilen zu können, wann eine kritische Anzahl von Interaktionspunkten vorliegt. Diese Daten sind domänenspezifisch zu erfassen, da die Art der Domäne einen starken Einfluss auf die Anzahl der Interaktionen haben wird (siehe die oben erwähnten Fallstudien). Daher würde eine solche Validierung verständlicherweise den Rahmen dieser Arbeit sprengen.

8.3.2 Einfache Bewertungstechniken

Neben solchen komplexen und damit nicht einfach zu berechnenden und validierenden Maßen kann man sich auch sehr einfache Messvorschriften konstruieren, die sich bereits mit einigen wenigen Zeile Code implementieren lassen.

Zur Illustration wollen wir hier lediglich ein äußerst einfaches Beispiel nennen, das der simplen Bestimmung der Komplexität eines Entwicklungsartefakts dient. Dabei soll die Komplexität C eines Requirements als die Summe der jeweils mit der Tiefe t der realizedBy-Ketten gewichteten Anzahl von Sub-Anforderungen bestimmt werden (t beginnt bei 1).

Im atomaren Artefakttyp Requirement lässt sich dieses Maß wie folgt berechnen:

```
1  /* Requirement: */
2  public int berechneC(int t) {
3      int k = t;
4      foreach(Task task in this.realizingTask)
5          k = k + task.berechneC(t+1);
6      return k;
7  }
```

Für unser *RoomAutomation*-Beispiel erhalten wir so z.B. die Messergebnisse aus Tabelle 8-3.

Tabelle 8-3. Komplexität der Anforderungen von *RoomAutomation*

Need	N1	N2	N3	N4
C	15	15	12	9

Auch hier müssen für den jeweiligen Zweck und eventuell die betrachtete Domäne die entsprechenden Maße gefunden und realisiert werden. Durch die Einfachheit einer solchen Umsetzung erscheint es allerdings plausibel, dass dies auch in realen Projekten gelingen wird und dort der Qualitätssicherung dienen kann.

An dieser Stelle wollen wir es mit der Illustration der Möglichkeiten, die sich durch unsere Automatisierungstechnik auch für die Berechnung von Maßen bietet, belassen. Eine Wiederaufnahme der Thematik der Messung von Artefakteigenschaften wird – wie bereits angedeutet wurde – in Abschnitt 10.1.1 auf Seite 302 erfolgen, wo wir den Beitrag einer Automatisierung quantifizieren werden.

8.4 Visualisierung von Modelleigenschaften

Sobald eine abstrakte Repräsentation der Modellinformation vorhanden ist (in unserem Falle die Instanziierung der abstrakten Artefakttypen), lässt sich daraus sehr einfach wieder eine konkrete Darstellung der Modellinformation erzeugen. Wir hatten dies in Abschnitt 7.2 auf Seite 180 für das Unparsen von PROBAnD-Dokumenten vorgestellt.

Neben der Erzeugung von Dokumenten in der konkreten Syntax der zu Grunde liegenden Entwicklungsmethode ist es oftmals aber auch hilfreich andere Sichten auf die Spezifikation zu erzeugen. Diese müssen – anders als die eigentlichen Entwicklungsdokumente – nicht unbedingt modifizierbar sein. Alleine die Visualisierung einzelner Modelleigenschaften kann schon zum Verständnis der Spezifikation beitragen und deren Analyse oder Review durch einen Menschen erleichtern.

Im Folgenden wollen wir dazu drei Beispiele aufzeigen, vor allem auch, weil diese durch deren äußerst effiziente Implementierung die Mächtigkeit unseres Ansatzes unterstreichen.

8.4.1 Interaktive Tabellendarstellung

Wir hatten in dieser Arbeit bereits mehrfach gezeigt, wie man sämtliche Artefaktinstanzen einer Spezifikation aufsammeln und für die entsprechenden Analysen oder Aktivitäten einsetzen kann (Marshalling, Konsistenzprüfung, etc.).

Eine besonders praktische Anwendung dieser Technik findet sich in der Darstellung der Modellinformationen in interaktiven Tabellen, wodurch eine schnelle Navigation durch das gesamte Modell ermöglicht wird. Damit wird eine Fehlersuche oder auch eine Inspektion stark vereinfacht, da man nicht die Information über viele Dokumente hinweg verfolgen muss. So müsste man in den PROBAnD-Dokumenten beispielsweise die Dokumenttypen Needs, TaskList und ControlObjectType betrachten, um ein vollständiges Bild der Realisierung einer Anforderung zu erhalten.

Mit einem grafischen *Browser* ist dies erheblich einfacher. Abb. 8-7 zeigt die Navigationsschritte in einem solchen Werkzeug. Ausgehend von Need N4 unseres *RoomAutomation*-Beispiels finden wir so z. B. die Realisierung durch Task T7 und dessen Implementierung im Objekttyp TempCtrl.

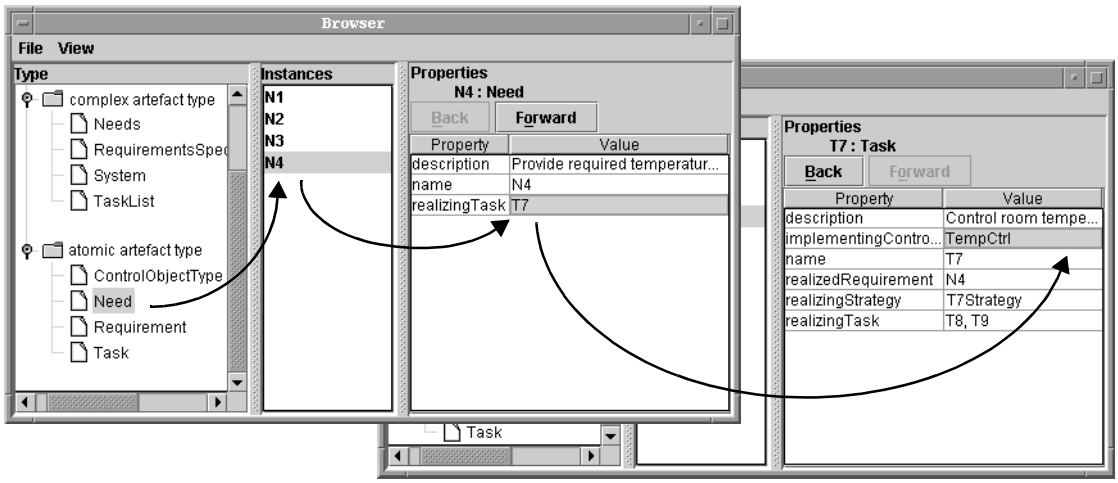


Abbildung 8-7. Anwendung des Browsers

Da die wesentlichen Aspekte der Implementierung eines solchen Werkzeuges bereits besprochen wurden und lediglich eine zusätzliche grafische Aufbereitung der Modellinformationen notwendig ist, wollen wir es bei dem Beispiel belassen.

8.4.2 Graphvisualisierung

Viele der Modellinformationen lassen sich als gerichteter Graph visualisieren, wobei i. d. R. die Knoten des Graphen die Artefaktinstanzen und die Kanten die Links zwischen diesen Instanzen repräsentieren.

Für die Erzeugung der grafischen Darstellung solcher gerichteter Graphen nutzten wir in unserer Arbeit das freie Graphvisualisierungswerkzeug *dot* des *Graphviz*-Projekts von AT&T. Als Eingabe erhält dot eine Datei mit einer sehr einfachen Syntax, welche die einzelnen Knoten und Kanten des Graphen spezifiziert. In Abb. 8-8 ist ein kleines Beispiel einer solchen Datei und der erzeugten Darstellung gezeigt.

Zunächst wird die Standardform der Knoten (`node`) spezifiziert und dann die Orientierung des Graphen (`TB` steht für „top-to-bottom“) angegeben. Als Nächstes folgt eine Liste der Knoten und zum Schluss die Angabe der diese Knoten verbindenden Kanten. Eine tiefergehende Einführung in das Werkzeug geben Gansner et al. in [GaN00].

Als Anwendung einer solchen Graphvisualisierung wollen wir im Folgenden zwei Beispiele aus dem Kontext der PROBAnD-Methode vorstellen.

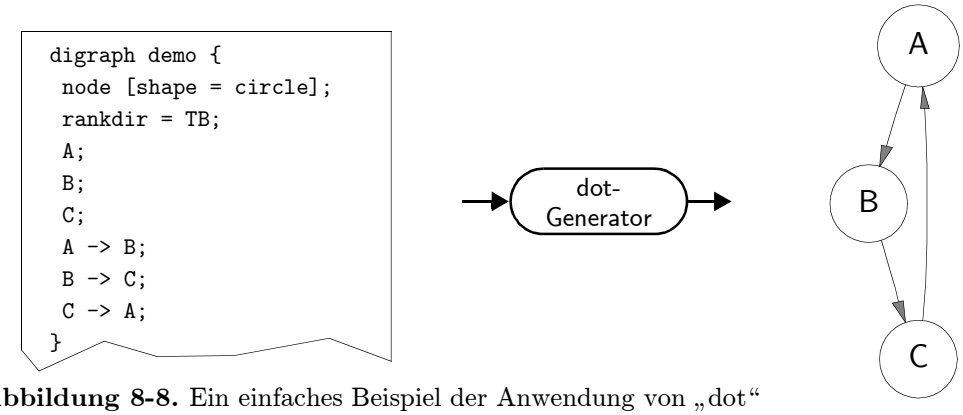


Abbildung 8-8. Ein einfaches Beispiel der Anwendung von „dot“

Visualisierung von Anforderungsgraphen

Bei der Vorstellung der automatischen Detektion von Feature-Interaktionen wurde in Abb. 8-2 auf Seite 210 ein sehr einfacher Anforderungsgraph gezeigt. Diesen konnten wir wegen der geringen Zahl an Anforderungen und realizedBy-Relationen noch von Hand erstellen. Für die Spezifikationen der Fallstudien *FloorAutomation* und *FloorAutomationX* erreicht die Komplexität eines solchen Graphen mit 331 Anforderungen und 298 realizedBy-Relationen bereits eine solche Komplexität, dass eine manuelle Erstellung nicht mehr sinnvoll möglich ist. Der in Abb. 8-6 auf Seite 228 gezeigte Graph war konsequenterweise automatisch aus der Modellinformation generiert worden.

Trotz der hohen Komplexität von Anforderungsgraphen für große Systeme – wie dem *FloorAutomationX*-System –, welche eine manuelle Detektion von Interaktionen erschwert, kann man an Hand der Darstellung dennoch sofort erkennen, welche Anforderungen nicht realisiert wurden, oder wo sich getrennte Anforderungsbereiche, die in einem vollständigen System eventuell verbunden sein müssten, befinden.

Für die Erzeugung einer Datei, aus welcher dann mit dem oben vorgestellten dot-Werkzeug ein Anforderungsgraph generiert werden kann, definieren wir eine Methode `createRequirementsGraph()`, wie folgt definiert:

```
1   public void createRequirementsGraph(Writer out,
2    RequirementsSpecification rs, Set pointsOfInteraction) {
3       out.println("digraph requirements {");
4       out.println("  node [shape = circle];");
5       out.println("  rankdir = TB;");
6       Set allRequirements = new HashSet();
7       foreach(Need need in rs.itsNeeds.itsNeed) {
8           out.println("  "+need.name+";");
9           allRequirements.add(need);
10      }
11      foreach(Task task in rs.itsTaskList.itsTask) {
```

```
12          if(pointsOfInteraction.contains(task)) {
13              out.println("  "+task.name+" [ style=filled ];");
14          } else {
15              out.println("  "+task.name+";");
16          }
17          allRequirements.add(task);
18      }
19      foreach(Requirement req in allRequirements) {
20          foreach(Task task in req.realizingTask) {
21              out.println("  "+req.name+" -> "+task.name+";");
22          }
23      }
24      out.println("}");
25  }
```

Zur Vereinfachung des Algorithmus gehen wir von der Annahme aus, dass alle Requirements unterschiedlich benannt sind (d.h. dass es keinen Need und keinen Task mit demselben Namen gibt). Damit können wir die Knoten des Graphen eindeutig mit den Namen der Anforderungen versehen.

Nach dem notwendigen Kopf der dot-Datei (siehe Abb. 8-8) folgen zuerst alle Needs und dann alle Tasks der Spezifikation. Dabei färben wir diejenigen Tasks, welche an einem Interaktionspunkt liegen, grau ein (Zeile 13).

Während der Erzeugung der Zeilen für die Knoten wurden gleichzeitig alle Anforderungen in der Menge `allRequirements` aufgesammelt. Daher können wir jetzt darüber iterieren und für jede Anforderung die Kanten zu den realisierten Tasks erzeugen.

Abb. 8-9 zeigt den automatisch generierten Graphen für unser *RoomAutomation*-System.

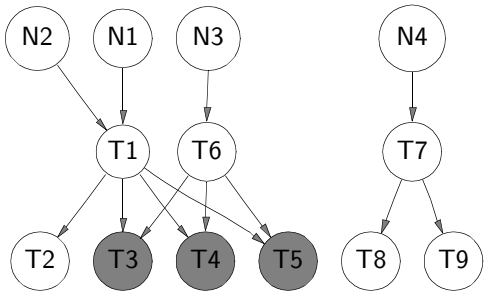


Abbildung 8-9. Automatisch generierter Anforderungsgraph

Visualisierung von Zustandsautomaten

Aus der operationalen Verhaltensbeschreibung der PROBAnD-Dokumente (siehe Abschnitt 6.2.3 auf Seite 151) werden i.d.R. SDL-Prozessgraphen erzeugt, auf welchen dann eine weitere Verfeinerung des Verhaltens vorgenommen werden kann.

Der Nachteil dieser SDL-„Zustandsautomaten“ ist allerdings, dass eine solche Spezifikation oft über viele Seiten verteilt ist und man nun schwer einen Gesamteindruck des Automaten gewinnen kann.

Mit der Möglichkeit einer automatischen Visualisierung kann man in unseren Ansatz eine klassische Zustandsautomatendarstellung (wie z. B. die der UML Statecharts) erzeugen, welche den obigen Mangel beseitigt. Diese kann dann für Dokumentations- und Inspektionszwecke eingesetzt werden. Auch hier können wir wieder auf das Werkzeug dot zurückgreifen, da es sich bei den Zustandsübergangsdiagrammen um gerichtete Graphen handelt, bei welchen die Knoten die Zustände und die Kanten die Transitionen darstellen. Abb. 8-10 zeigt den automatisch generierten Statechart für das TempCtrl-Beispiel aus Tabelle 6-5 auf Seite 142.

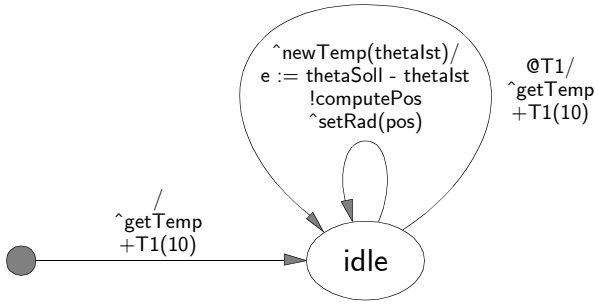


Abbildung 8-10. Automatisch generierter Zustandsübergangsgraph für TempCtrl Objekttyp

Wichtig für eine solche Diagrammdarstellung ist, dass man die „*“- und die „–“-Zustände von SDL auflöst, da diese in den normalen Zustandsübergangsdiagrammen nicht verwendet werden. Die UML Statecharts bieten zwar die Möglichkeit mit hierarchischen Zuständen einen ähnlichen Sachverhalt zu modellieren (eine Transition ausgehend von einem Stern-Zustand kann in den Statecharts als eine Transition zwischen den „Ober“-Zuständen modelliert werden, siehe dazu z. B. [HrR02, S. 78]), allerdings funktioniert dies nicht in allen Fällen. Insbesondere wenn mehrere „*“-Zustände in SDL auftauchen, die nicht überlappungsfrei sind (weil sie unterschiedliche Zustände ausschließen), lässt sich eine entsprechende Statechart-Darstellung nur schwer ermitteln.

Die Erzeugung eines „flachen“ Zustandsdiagramm in der `createStateGraph()`-Operation beginnt daher zunächst mit der Bestimmung aller Zustände und der Erzeugung der entsprechenden Knoten im Graphen:

```
1   public void createStateGraph(ControlObjectTypeConfiguration cotc,
2    Writer out) {
3       out.println("digraph statechart {");
4       out.println("  node [shape = ellipse];");
5       out.println("  rankdir = LR;");
6
```

```
7       Set allStates = new HashSet();
8       State startState = new State("start");
9       allStates.add(startState);
10      AbstractStateSet ass;
11      foreach(Transition tr in cotc.itsTransition) {
12          ass = tr.originatingAbstractStateSet;
13          if(ass.type == StateSet) {
14              StateSet ss = (StateSet)ass;
15              allStates.addAll(ss.includedState);
16          }
17      }
18
19      foreach(State st in allStates) {
20          if(st == startState) {
21              out.println("  "+st+" [ label = \"\","+
22               " shape=circle, style=filled ];");
23          } else {
24              out.println("  "+st+";");
25          }
26      }
```

Zum Anfang der Methode wird der bereits bekannte dot-Kopf erzeugt. Diesmal erfolgt die Graph-Darstellung allerdings von links nach rechts (`rankdir = LR`).

Nun müssen alle Zustände, die in dem spezifizierten Objekttyp verwendet werden, aufgesammelt werden. Wir beginnen dazu mit der Definition des „Startsymbols“ (`startState`), welches in den PROBAnD-Dokumenten nicht explizit spezifiziert wird, in unserem Graph aber als Knoten auftauchen soll.

Danach iterieren wir über alle Transitionen des Objekttyps und fügen alle Zustände der Ausgangszustandsmenge (`StateSet`) zu der Menge aller Zustände (`allStates`) hinzu. Damit erhalten wir alle Zustände, da wir in Constraint $CO1$ des Abschnitt 8.1.2 auf Seite 192 gefordert hatten, dass alle Zustände des Automaten mindestens einmal als Ausgangszustand auftauchen müssen.

Diese Zustände werden nun als Knoten ausgegeben, wobei wir für das „Startsymbol“ einen gefüllten Kreis wählen. Als Nächstes folgt die Erzeugung der Transitionen (also der Kanten des Graphen):

```
27      foreach(Transition tr in cotc.itsTransition) {
28          ass = tr.originatingAbstractStateSet;
29          if(ass == null) {
30              StateSet newss = new StateSet("");
31              newss.includedState += startState;
32              ass = newss;
33          }
```

```
34          termState = tr.terminatingState;
35          String transDescr = buildTransString(tr);
36          if(ass.type == StateSet) {
37              StateSet ss = (StateSet)ass;
38              foreach(State origState in ss.includedState) {
39                  State destState;
40                  if(termState.name.equals("-")) {
41                      destState = origState;
42                  } else {
43                      destState = termState;
44                  }
45                  out.println("  "+origState.name+" -> "+destState.name+
46                   " [ label = \""+transDescr+"\" ];");
47              }
48          } else {
49              iss = (InverseStateSet)ass;
50              Set excludedStates = iss.excludedState;
51              Set remainingStates = new HashSet(allStates);
52              remainingStates.removeAll(excludedStates);
53              remainingStates.remove(startState);
54              foreach(State origState in remainingStates) {
55                  State destState;
56                  if(destState.name.equals("-")) {
57                      destState = origState;
58                  } else {
59                      destState = termState;
60                  }
61                  out.println("  "+origState.name+" -> "+destState.name+
62                   " [ label = \""+transDescr+"\" ];");
63              }
64          }
65      }
66      out.println("}");
67  }
```

Für jede Transition wird im obigen Code eine Kante im Graphen mit der entsprechenden Beschriftung erzeugt. Zunächst prüfen wir, ob es sich um eine Initialisierungstransition handelt, also um eine Transition, welche keinen Ausgangszustand besitzt. In diesem Fall erzeugen wir eine „hypothetische“ Zustandsmenge mit dem weiter oben definierten `startState`. Dadurch können wir die nun folgende Kantenerzeugung sehr allgemein kodieren.

In Zeile 35 wird die Beschriftung der Transition berechnet, die dem Syntax der HTML-Transitionen entspricht, bei welchen die Ausgangs- und Zielzustände weg-

gelassen werden (vgl. Abschnitt 6.2.3 auf Seite 151). Wir wollen die Methode `buildTransString()` hier nicht erläutern, da deren Implementierung analog zu der Konstruktion der konkreten Artefakte ist, wie sie in Abschnitt 7.2 auf Seite 180 skizziert wurde.

Ist die Ausgangszustandsmenge vom Typ StateSet, so müssen für jeden der Zustände dieser Menge eine entsprechende Transition erzeugt werden (Zeilen 36 bis 47). Handelt es sich bei dieser Menge um einen InverseStateSet (d. h. einen „*“-State in SDL), so müssen entsprechende Transitionen für alle außer den in der Menge ausgeschlossenen Zuständen (`excludedState`) des Objekttyps generiert werden (Zeilen 49 bis 63). Dazu kopiert man zunächst die Menge aller Zustände und entfernt aus dieser Menge (`remainingStates`) dann alle exkludierten Zustände und den `startState`.

In beiden Fällen wird vor der Erzeugung der eigentlichen Kante noch geprüft, ob es sich bei dem Folgezustand um einen „–“-Zustand handelt. In einem solchen Fall wird als Folgezustand wieder der Ausgangszustand (`origState`) gewählt.

Zusammenfassung

Eine Software-Spezifikation besitzt eine hohe interne Qualität, falls die Dokumente sowohl konsistent (ohne Widersprüche oder Mehrdeutigkeiten) als auch vollständig (ohne fehlende Informationen) sind, falls keine unerwünschten Interaktionen zwischen Anforderungen existieren und falls die geforderten (quantitativen) Qualitätsmerkmale (z. B. Komplexität einzelner Artefakte) eingehalten werden.

Für alle diese Aspekte wurden in diesem Kapitel Strategien zur maschinellen Unterstützung durch Werkzeuge vorgestellt und deren Implementierung auf Basis unseres verbesserten Metamodellansatzes und unserer Aktionssprache AL++ erläutert.

Schließlich wurde eine mögliche Visualisierung von Modelleigenschaften diskutiert und an drei Beispielen illustriert.

Da bisher kein AL++-Compiler (oder eine entsprechende Modellierungsumgebung, vgl. Abschnitt 11.2.1 auf Seite 343) existiert, wurde zur ablauffähigen Realisierung der hier vorgestellten Algorithmen die Abbildung aus Abschnitt 5.2 auf Seite 96 angewendet.

9 Automatisiertes Prototyping für reaktive Systeme

As The Standish Group discovered, the typical development project only includes 54 percent of the originally defined features. But what's even more disturbing is that customers typically use only 55 percent of the features that are delivered. Why? Quite often, the problem is due to poorly defined requirements.

— [Telelogic „Automate" Bulletin, 3(11), November 2003]

Im Rahmen der Qualitätssicherung dient Prototyping sowohl der Sicherstellung der externen Produktqualität, da wichtige externe Eigenschaften des Systems sich bereits am Prototypen beobachten lassen, als auch der Sicherstellung der internen Produktqualität, was durch Tests ermöglicht wird.

Bereits 1979 förderte eine US-Studie zu Tage, dass das Versagen vieler Software-Projekte an der unzulänglichen Anforderungsbeschreibung der Nutzer lag. Als Lösung dieses Problems kristallisierten sich in den 1980er Jahren zwei Ansätze heraus: eine Formalisierung der frühen Aktivitäten, um fehlerhafte Anforderungen formal identifizieren zu können, und ein experimenteller Ansatz, der den Erfahrungsgewinn unterstützt. Prototyping ist in diesem Sinne dem zweiten Ansatz zuzuordnen [BKK92, S.6].

9.1 Grundlagen des Prototypings

Unter dem Begriff „*Prototyp*" wird in dieser Arbeit stets ein Software-Prototyp verstanden, welcher in der folgenden Definition (nach [PPS92, S.50]) präzisiert wird:

Definition 9-1: Ein **Software-Prototyp** ist ein ausführbares Artefakt mit den wesentlichen Eigenschaften des Zielsystems, das die Kommunikation zwischen Kunden und Entwicklern unterstützt. Der Software-Prototyp bildet wesentliche Eigenschaften des geplanten Systems in einer anschaulichen und leicht modifizierbaren Form nach. Ein Prototyp soll die Funktionalität wirkungsvoller darstellen als textuelle Beschreibungen oder rein statische (= nicht ausführbare) Artefakte.

Der Begriff des *Prototypings* schließt dabei sowohl die Aktivitäten der Erstellung des Prototyps (Prototyping i.e.S.) als auch dessen Anwendung (Prototyping i.w.S.) mit ein. Auf diese Aktivitäten werden wir im Folgenden näher eingehen.

9.1.1 Aktivitäten des Prototypings

Floyd unterscheidet in [Flo84] vier Aktivitäten des Prototypings (von der Autorin Phasen genannt): *funktionale Selektion*, *Konstruktion*, *Evaluierung* und *weitere Verwendung*. In Abb. 9-1 sind die Aktivitäten Selektion, Konstruktion und Evaluierung im Kontext einer abstrakten Sicht auf die Software-Entwicklung dargestellt.

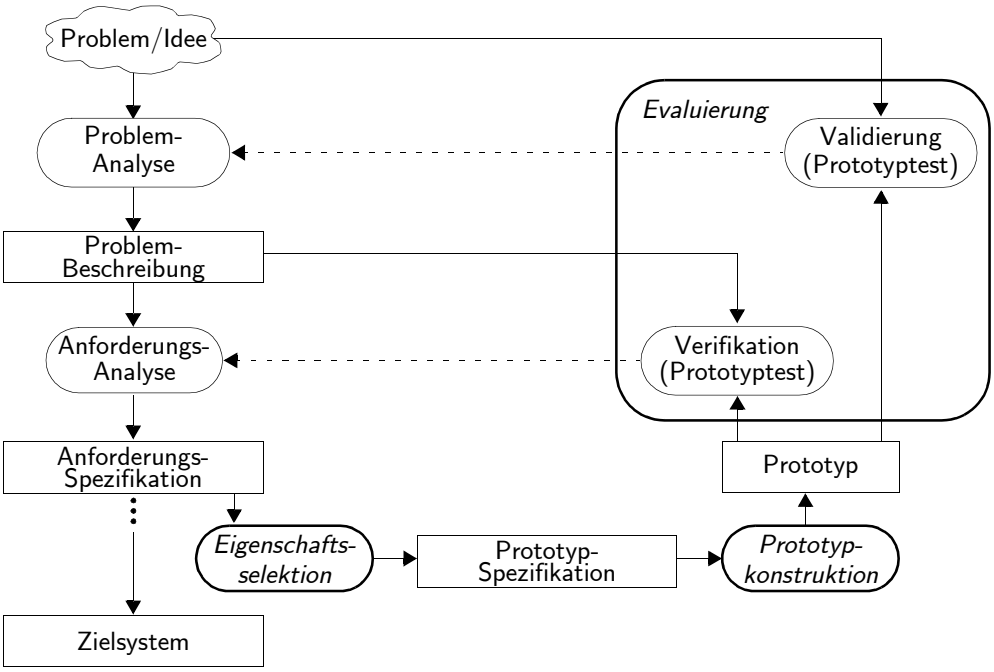


Abbildung 9-1. Prototyping in der Software-Entwicklung

Ausgehend von einer *Problembeschreibung*, die während der *Problemanalyse* niedergeschrieben wird, erfolgt in der *Anforderungsanalyse* die Formulierung der *Anforderungsspezifikation*. In einem Software-Entwicklungsprozess ohne Prototyping würde die Systementwicklung nun auf traditionelle Weise bis hin zum *Zielsystem*

erfolgen. Durch Prototyping lässt sich dieser Weg zu einem ausführbaren Produkt zunächst abkürzen.

Selektion der relevanten Eigenschaften

Wie aus Definition 9-1 ersichtlich ist, besitzt ein Prototyp die jeweils wesentlichen oder relevanten Eigenschaften des Zielsystems. Ein erster Schritt im Prototyping ist daher die Auswahl der gewünschten Eigenschaften. Anders als Floyd denken wir, dass eine Erweiterung der Selektion auf alle (auch die nicht-funktionalen) Eigenschaften des Zielsystems angebracht ist. So kann es z.B. bereits beim Prototyping eines ABS-Steuerungssystems notwendig sein, Zeiteigenschaften zu berücksichtigen.

Eine Selektion kann unter verschiedenen Gesichtspunkten vorgenommen werden, s.d. sich Prototyp und Zielsystems bzgl. deren Eigenschaften wie folgt unterscheiden können:

1. Die Eigenschaften, die im Prototyp realisiert werden, bilden eine Teilmenge der Eigenschaften des Zielsystems. Die realisierten Eigenschaften werden dabei in ihrer endgültigen Form und Präzision implementiert (*vertikales Prototyping*).
2. Alle Eigenschaften des Zielsystems sind auch im Prototyp zu finden. Allerdings ist der Detaillierungsgrad der Implementierung grober (oder abstrakter) als im Zielsystem. Bestimmte Effekte werden übergangen oder simuliert (*horizontales Prototyping*).
3. Es liegt eine Kombination der ersten beiden Arten vor (*diagonales Prototyping*).

Für die weiteren Betrachtungen wollen wir davon ausgehen, dass durch diese Selektionsaktivität zunächst eine Spezifikation des Prototyps entsteht (siehe Abb. 9-1), welche sich insbesondere beim vertikalen Prototyping stark von der Anforderungsspezifikation unterscheiden kann (sie ist ein Ausschnitt aus dieser). Wie diese Selektion durch unseren modellbasierten Automatisierungsansatz unterstützt werden kann, wird in Abschnitt 9.2.1 auf Seite 258 und Abschnitt 9.2.2 auf Seite 266 erläutert.

Konstruktion

Ein wichtiges Kriterium für die Akzeptanz des Prototypings ist, dass die Erstellung des Prototyps mit vertretbarem Aufwand möglich ist, d.h. der Aufwand für die Erstellung (Implementierung) der Prototypen sollte (wesentlich) geringer als der Aufwand zur Konstruktion des eigentlichen Systems sein.

Die eben beschriebene Auswahl relevanter Eigenschaften leistet hier bereits einen wichtigen Beitrag. Zu den Eigenschaften, die häufig vernachlässigt werden können,

zählen insbesondere Qualitätsmerkmale, die für langlebige Produkte wichtig sind, wie z.B. Verlässlichkeit, Robustheit, Effizienz oder auch Wartbarkeit.

Durch den Einsatz angemessener Techniken und Werkzeuge bei der Konstruktion der Prototypen lässt sich dieser Aufwand weiter reduzieren. Die Implementierung eines Prototyps kann dabei prinzipiell auf zwei Arten erfolgen: manuell oder automatisiert.

Ein erheblicher Effizienzgewinn lässt sich offensichtlich durch eine automatische Prototypgenerierung erreichen, welche wir in Abschnitt 9.2 auf Seite 258 vorstellen werden. Dazu muss allerdings das Ausgangsmodell (die Prototypspezifikation), aus welchem der Prototyp abgeleitet wird, eine formale Syntax besitzen (siehe dazu Abschnitt 2.1.1 auf Seite 11).

Evaluierung

Die Ausführbarkeit der Prototypen ermöglicht deren Anwendbarkeit für das Testen. Daher kann Prototyping sowohl für die *Validierung* als auch für die *Verifikation* eingesetzt werden. Beide Begriffe werden in der Literatur leider nicht konsistent verwendet, weshalb wir die nun vorgestellte Definition im Verlauf der Arbeit verwenden werden.

Eine Validierung soll die Frage „wird das *richtige* System entwickelt?“ beantworten (siehe dazu auch z.B. [BrH93, S.334]) und trägt somit zur Sicherstellung des Qualitätsmerkmals „Zuverlässigkeit“ bei (siehe Abschnitt 7.1.1 auf Seite 177). Um dies zu erreichen, muss sich der Benutzer einen Eindruck vom Produkt verschaffen können, um es gegen seine Vorstellungen (*nicht* die Problembeschreibung!) überprüfen zu können (siehe Abb. 9-1). In der Regel erfolgt dies am Ende der Entwicklung mit Hilfe des Zielsystems. Man spricht dann von einem *Produkttest*. Werden Prototypen eingesetzt, so kann diese Validierung bereits früher im Entwicklungsablauf erfolgen. Diese *Prototyptests* können daher zu einer Verkürzung des Validierungszyklus beitragen. In Abschnitt 9.3 auf Seite 275 werden wir zeigen, wie man solche Prototyptests und weitergehende Prototyping-Aktivitäten durch eine Automatisierung unterstützen kann.

Sobald ein ausführbares System zur Verfügung steht, kann es von den Entwicklern auch zur Verifikation eingesetzt werden, also zur Beantwortung der Frage „wird das System *richtig* entwickelt?“. Dazu ist jeweils zu prüfen, ob der getestete Teil des Systems auch seiner Spezifikation entspricht (in Abb. 9-1 ist z.B. zu sehen, dass die Anforderungsspezifikation gegen die Problembeschreibung verifiziert wird). Neben einfachen Tests sind auf der Basis von Prototypen auch komplexe dynamische Analysen möglich, auf welche wir in Abschnitt 9.3.2 auf Seite 282 eingehen werden. Durch die Verifikation wird die Sicherstellung des Qualitätsmerkmals „Korrektheit“ erreicht (vgl. Abschnitt 7.1.1 auf Seite 177).

Durch Prototyptests können viele Fehler früh in der Entwicklung gefunden und korrigiert werden (gestrichelte Linien in Abb. 9-1 auf Seite 242), deren Korrektur in späteren Entwicklungsstadien aufwändiger wäre (siehe auch Abschnitt 7.1.1 auf Seite 177). Konsequenterweise sollte man den Einsatz von Prototypen so früh wie möglich in der Entwicklung erlauben. In diesem Zusammenhang findet man häufig auch den Begriff „*Rapid Prototyping*“, der als die schnelle und günstige Erstellung eines voll funktionsfähigen Prototypen bereits sehr früh in der Entwicklung verstanden werden kann (siehe z.B. [Dew94]). Allerdings taucht dieser Begriff auch sehr oft im Bereich des Hardware-Software-Co-Designs auf (vgl. z.B. [KoA03]), weshalb wir den Ausdruck „frühes Prototyping“ bevorzugen.

Weitere Verwendung

Die weitere Verwendung des Prototyps hängt vom Zweck des Prototypings ab:

1. *Exploratives Prototyping*: Ein Prototyp ist stets ein wichtiges Kommunikationsmittel zwischen Entwicklern und Kunden. Beim explorativen Prototyping wird eine Vorführung potenzieller Systemeigenschaften genutzt, um dem Kunden die Auswahl der gewünschten Systemfunktionalität zu erleichtern. Auch eine bewusst „schlechte“ Lösung kann hierbei helfen die Grenzen der Benutzeranforderungen auszuloten. Nach der Erstellung der Anforderungsspezifikation wird der Prototyp „weggeworfen“ („throw-away prototype“).
2. *Experimentelles Prototyping*: Nachdem eine initiale Anforderungsspezifikation erstellt wurde, kann der Benutzer mit einem daraus abgeleiteten Prototyp erste Experimente – möglichst in einem realen Kontext – durchführen. Oftmals wird daher ein solcher Prototyp als Ergänzung (oder sogar Verfeinerung) der Anforderungsspezifikation behandelt (in Abschnitt 9.1.2 werden wir auf diesen Aspekt unter anderen Gesichtspunkten nochmals eingehen). Beim experimentellen Prototyping könnte der Prototyp inkrementell zum endgültigen Produkt weiterentwickelt werden. Allerdings birgt eine solche Systementwicklung auch die Gefahr, nicht-wartbaren Code zu erzeugen, da gerade die Qualitätseigenschaften, welche für langlebige Produkte relevant sind, nicht in den Prototyp eingeflossen sind. Einzig sinnvolle Verwendung des Prototyps kann also wiederum nur sein, diesen „wegzuwerfen“.
3. *Evolutionäres Prototyping*: Auf der Grundlage der Erkenntnis, dass sich durch die Einführung eines Software-Systems die Anforderungen an solch ein System ändern („requirements creep“ [CAD01]), führt man mit dem evolutionären Prototyping eine Entwicklung in Versionen ein. Durch jeweils neue Entwicklungszyklen (Erstellung einer neuen Version) lässt sich auf geänderte Anforderungen reagieren. Unabdingbare Voraussetzung für eine solche Vorgehensweise ist allerdings, dass Eigenschaften wie z.B. Wartbarkeit im Prototyp Berücksichtigung finden.

4. *Inkrementelles Prototyping*: Ausgehend von einer vollständigen Anforderungsspezifikation wird das Zielsystem schrittweise in einzelnen Inkrementen (Tranchen) entwickelt. Damit gliedert sich das inkrementelle Prototyping nahtlos in ein inkrementelles Vorgehensmodell ein.

Eine detailliertere Behandlung dieser Aspekte, die auf den Ergebnissen von Floyd [Flo84] und Hekmatpour et al. [HeI88] basieren, findet man in [Met98].

9.1.2 Unterschied zwischen Spezifikation und Prototyp

Sehr wichtig ist in unseren Augen eine klare Unterscheidung der Begriffe „Spezifikation“ und „Prototyp“. Auch Queins weist in seiner Arbeit auf die Unterschiede dieser Artefakte hin [Que02, S. 175ff.]. Eine solche Unterscheidung ist insbesondere deshalb relevant, da die Annahme, dass die Eigenschaften des Prototyps und dessen Spezifikation identisch sind, bei einer Verifikation zu einer fälschlichen Identifikation von Fehlern in der Spezifikation führen kann, oder – was schlimmer ist – Fehler auch verdecken kann. Die Ursachen dieses Problems liegen in der Abbildung, die vorgenommen wird, um aus einer Anforderungsspezifikation einen Prototyp zu erzeugen.

Neben dem offensichtlichen Unterschied zwischen Anforderungs- und Prototypspezifikation auf Grund der Eigenschaftsselektion kommt es auch durch die Konstruktion (Implementierung) eines Prototyps aus dessen Spezifikation zu Abweichungen.

Da es sich sowohl bei der Prototypspezifikation als auch bei dem Prototypen um Modelle handelt (beide abstrahieren mehr oder weniger von dem eigentlichen System), ist jede Prototypkonstruktion auch eine Modelltransformation. Da sich die Formalismen von Spezifikation und Prototyp allerdings i. d. R. deutlich unterscheiden (der Formalismus der Spezifikation bei der PROBAnD-Methode ist beispielsweise SDL wohingegen der Prototyp in C implementiert wird), impliziert dies eine gravierende Modelländerung. Nach unserer Klassifikation der Abbildungen aus Abschnitt 2.2.2 auf Seite 19 besteht zwischen der Spezifikation und dem Prototyp also eine Abbildung vom Typ MFa (echte Formalismus- und Modelländerung).

Idealerweise sollte das von der Prototypspezifikation beschriebene System und das letztlich durch den Prototyp realisierte System identisch sein, um an Hand des Prototyps die Verifikation der Spezifikation durchführen zu können. In der Klassifikation aus Kapitel 2 sollte also eine Abbildung vom Typ 0M existieren. Genau diese gewünschte Abbildung wird allerdings durch die notwendige Transformation vom Typ MFa erschwert, wenn nicht sogar ausgeschlossen, da sich die Formalismen normalerweise nicht nur in deren Syntax sondern auch in deren Semantik, also dem zu Grunde liegenden Paradigma der Konzepte (siehe auch [Hol03, S. 7f.] und [KlW03]), unterscheiden.

Bei der Abbildung einzelner Modellelemente der Spezifikation auf die des Prototyps lassen sich daher drei Fälle unterscheiden:

1. *Eindeutige Abbildung*: Ein oder mehrere Modellelemente der Spezifikation können bedeutungserhaltend und eindeutig auf ein oder mehrere Elemente des Prototyps abgebildet werden. Dieser Fall ist unproblematisch und würde zu einer Abbildung vom Typ 0M führen, wenn dies für alle Elemente der Spezifikation gelten würde.
2. *Auswahl*: Ein oder mehrere Elemente der Spezifikation lassen sich nur nach weiteren Festlegungen auf Elemente des Prototypformalimus abbilden. Diese Abbildung ist daher nicht eindeutig, was insbesondere bedeutet, dass eine Auswahl aus möglichen alternativen Elementen des Prototypformalismus getroffen werden muss.
3. *Approximation*: Für ein oder mehrere Elemente der Prototypspezifikation existiert keine Entsprechung in dem Formalismus des Prototyps. Es muss versucht werden, diese Elemente so gut es geht mit den Elementen des Prototypformalismus nachzuahmen, zu „approximieren".

Die letzten beiden Fälle sind diejenigen, die eine Abbildung vom Typ 0M verbieten. Denn können die Elemente der Spezifikation keine oder zumindest keine eindeutige Abbildung auf den Prototyp erfahren, so können die Systeme nur schwerlich identisch sein.

Wird allerdings eine automatisierte Prototypkonstruktion (also eine Generierung) eingesetzt, dann werden die Abbildungen der Modellelemente stets nach den im Generator fest-kodierten Regeln vorgenommen. Dies kann bei der Auswertung der Testläufe berücksichtigt werden und damit eine richtige Beurteilung der Ergebnisse der Verifikation getroffen werden (siehe auch [Que02, S.173]).

Wir wollen im Folgenden die Fälle 2 und 3 näher erläutern.

Auswahl (Fall 2)

Die Situation aus Fall 2 findet sich beispielsweise bei der Realisierung der parallelen Prozesse von SDL (siehe auch Abschnitt 6.1.3 auf Seite 138) durch eine Abbildung auf einen monolithischen Betriebssystemprozess, der in C implementiert wird (Ansatz von Telelogic in der Tau SDL Suite). Hierzu muss eine Reihenfolge der ausführbaren Prozesse festgelegt werden, um diese sequenziell abarbeiten zu können. Auf Grund dieser Sequenzialisierung kann es nun allerdings vorkommen, dass kritische Situationen im Prototypen nie auftreten und daher bestimmte Fehler in keinem der Prototyptests erkannt werden können.

Zur Veranschaulichung betrachten wir uns das SDL-Beispiel aus Abb. 9-2, in welchem neben der Architektur des SDL-Systems auch die Zustandsgraphen der Prozesse gezeigt sind.

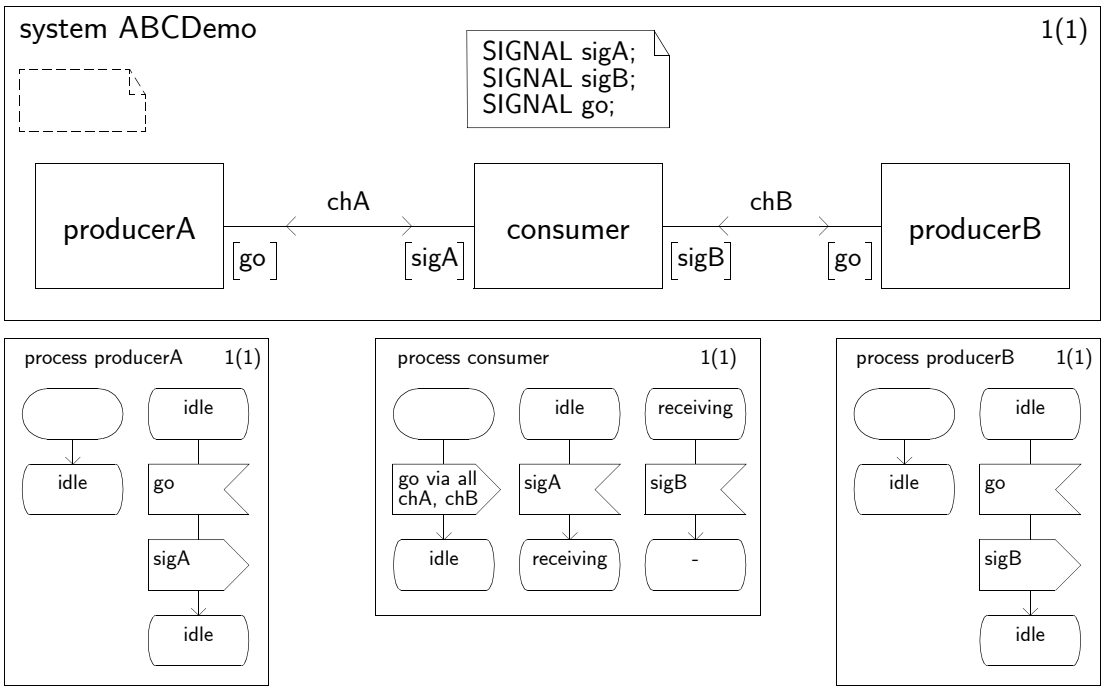


Abbildung 9-2. Beispielhaftes SDL-System

Dieses SDL-System beschreibt drei kommunizierende Prozesse, wobei der consumer-Prozess Signale von producerB erst dann empfangen kann, wenn producerA eine Instanz des Signals sigA an den consumer gesendet hat. Zum Start der Kommunikation sendet der consumer eine Instanz des go-Signals an die beiden produzierenden Prozesse. Je nachdem, ob nun producerA oder producerB zuerst aktiviert wird, kann der consumer die Signalinstanz sigB empfangen oder geht diese verloren. Abb. 9-3 zeigt die beiden alternativen Abarbeitungsreihenfolgen als *Message Sequence Chart* (*MSC* [OFM94, S.356ff.]; entspricht in der verwendeten Form weitgehend den UML-Sequenzdiagrammen).

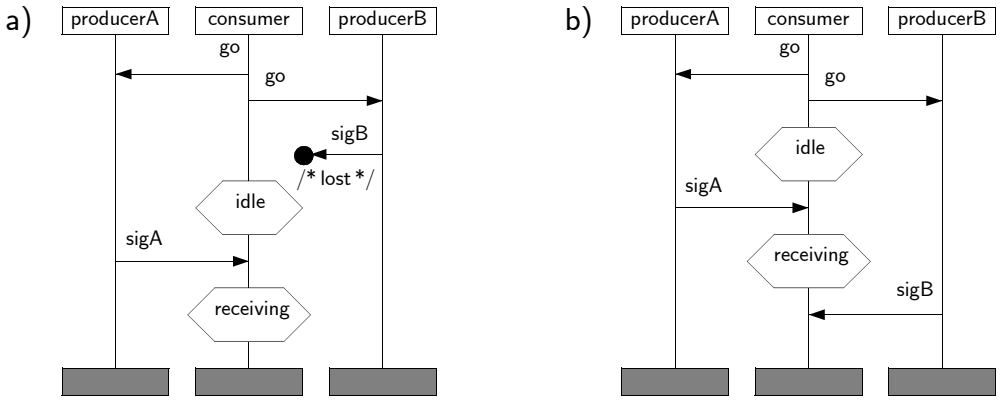


Abbildung 9-3. Alternative Abarbeitungsreihenfolgen des SDL-Systems aus Abb. 9-2

Werden die Prozesse immer nur nach dem Schema aus Szenario b) in Abb. 9-3 aktiviert, dann handelt es sich um eine Verdeckung von Fehlern in der Spezifikation, da hierbei der potenzielle Verlust der Signalistanzen vom Typ sigB nicht erkannt werden kann.

Man kann beim Prototyping nun versuchen für alle Elementtypen, für die eine Auswahl erfolgen muss, alle Alternativen zu testen, indem man bei jeder Abbildung der Prototypspezifikation auf den Prototyp eine andere Möglichkeit realisiert. Neben einer zum Teil schon unüberschaubaren Anzahl an Alternativen (bei der Prozessaktivierung müsste man alle möglichen Aktivierungsreihenfolgen aller Prozesse eines SDL-Systems betrachten), besteht sehr häufig bzgl. vieler Elementtypen der Spezifikation Wahlfreiheit in der Realisierung. So kann bei der Abbildung von SDL auch die Verzögerung von Signalen entlang von Kanälen unterschiedlich abgebildet werden. Daher kann die kombinatorische Vielfalt aller Alternativen sehr schnell Dimensionen erreichen, die das erschöpfende Testen in der Praxis nicht mehr erlaubt.

Man kann sich aber des Zufalls behelfen, um eine Teilmenge aller Möglichkeiten zu selektieren und nur diese zu prüfen. Insbesondere da man durch Testen im Allgemeinen nicht in die Lage versetzt wird, die Korrektheit zu beweisen, liefert das Testen von mehreren Arten der Abbildung zumindest eine Erhöhung der Konfidenz, ein korrektes Produkt zu entwickeln.

Da wir für unsere Arbeiten von einem gegebenen Code-Generator (Telelogic Tau) für die Abbildung von SDL nach C-Code ausgehen (siehe auch Abschnitt 9.2 auf Seite 258), können wir diesen Schritt allerdings nicht durch unseren Automatisierungsansatz behandeln. Die Telelogic Tau Umgebung beinhaltet aber zumindest für das obige Problem der Aktivierungsreihenfolge entsprechende Analysewerkzeuge, die z.B. einen zufälligen Lauf („Random Simulation“, siehe [Dol03]) oder die Analyse einer echt parallelen Abarbeitung erlauben.

Approximation (Fall 3)

Ein typisches Beispiel für den dritten Fall ist die Realisierung der unendlichen Signalwarteschlangen von SDL (siehe Abschnitt 6.2.3 auf Seite 151). Diese können auf Grund der Speicherbegrenzung eines realen Rechners nur auf Datenstrukturen einer endlichen Länge abgebildet werden. Insbesondere bei eingebetteten Systemen mit geringer Speichergröße können daher Speicherüberlaufsfehler während des Prototypings auftreten, die nicht auf einen Fehler in der Spezifikation zurückzuführen sind. Queins stellt in [Que02, S. 179ff.] dieses Beispiel ausführlicher dar.

Zusätzliche Eigenschaften

Zusätzlich zu den in der Spezifikation (und dadurch auch im Prototyp) beschriebenen Eigenschaften können an einem ausführbaren System zusätzliche Eigenschaften

beobachtet werden. Dies ist in der Abstraktion begründet, die ein Modell bzw. ein Prototyp gegenüber dem modellierten System aufweist. So wird z.B. die konkrete Ausführungszeit ein und desselben Systems je nach Auslastung des Host-Rechners unterschiedlich sein. Hierbei handelt es sich also um eine Abbildung vom Typ S0 (siehe Abschnitt 2.2.2 auf Seite 19).

Zusammen mit den weiter oben angeführten Unterschieden zwischen Spezifikation und Prototyp, liefert die im letzten Absatz erläuterte Tatsache eine Begründung dafür, weshalb Prototypen, die von einem Benutzer oder Kunden validiert und akzeptiert wurden, auch als Teil der Anforderungsspezifikation betrachtet werden sollten. Auch Budde at al. weisen in [BKK92, S.40] auf eine solche Verwendung von Prototypen hin, ohne jedoch den Grund für diese Vorgehensweise zu beleuchten.

9.1.3 Prototyping für reaktive Systeme

Reaktive Systeme weisen besondere Eigenschaften auf, die beim Prototyping berücksichtigt werden müssen. Die vorstechendste Eigenschaft ist die bereits in Abschnitt 6.1.1 auf Seite 132 erläuterte Einbettung in eine physikalische Umgebung. Daneben kann aber auch das Zeitverhalten der Systeme für das Prototyping relevant sein.

Vereinfacht wird das Prototyping reaktiver Systeme auf Grund der Tatsache, dass diese über einen sehr geringen Anteil grafischer Benutzeroberflächen (GUI) verfügen, was beim Prototyping von Transformations- bzw. Dialogsystemen einen bedeutenden Anteil einnimmt (siehe dazu z.B. [BiP92, S.33]).

Einbettung

Für realistische Tests mit einem reaktiven Prototypen (was insbesondere für die Validierung relevant ist) ist die Berücksichtigung der Umgebung des reaktiven Systems unabdingbar (vgl. auch [Tra93, S.148]). Diese Umgebung liefert Stimuli (Messgrößen der Sensoren), auf welche das reaktive System durch die Erzeugung entsprechender Stellgrößen (für Aktuatoren) reagiert. Durch diese Stellgrößen wird die Umgebung beeinflusst, was wiederum zu einer Beeinflussung der Messgrößen führt.

Da diese Rückkopplungen bei komplexen Systemen nicht mehr sinnvoll statisch beschrieben werden können (z.B. in Tabellen), ist die Existenz der Umgebung für realistische Tests unabdingbar. Insbesondere werden Umgebungssimulatoren für ein frühes Prototyping notwendig, da zu Beginn der Entwicklung nicht davon ausgegangen werden kann, dass die physikalische Umgebung (z.B. eine Gebäude) bereits existiert. Desweiteren können mit Hilfe von Simulatoren weitergehende Tests als mit der realen Umgebung gemacht werden. So können z.B. Extremwerte getestet

oder in der Realität sehr selten auftretende Situationen evaluiert werden (siehe dazu auch [MMZ02]).

Diese Simulatoren müssen aber nicht notwendigerweise die Gesamtumgebung nachbilden, es genügt i. d. R. dass diese den technischen Prozess (also die beeinflusste Umgebung, siehe dazu Abb. 6-1 auf Seite 133) simulieren können. Insbesondere die Beeinflussung der Hardware-Komponenten des reaktiven Systems durch die umfassendere Umgebung kann vernachlässigt werden. Die Rückwirkung des technischen Prozesses auf die umfassendere Umgebung findet entweder nicht statt (z. B. wird das Wetter nicht durch eine Heizungsregelung beeinträchtigt) oder ist vernachlässigbar (die „Aufheizung" der Außenluft durch ein beheiztes Gebäude ist marginal). Daher kann der Einfluss der umfassenderen Umgebung auf den technischen Prozess statisch (z.B. durch Wetter-Files) beschrieben werden.

Die enge Kopplung von reaktivem System und technischem Prozess erfordert kompatible Schnittstellen auf beiden Seiten. Wir werden in Abschnitt 9.2.3 auf Seite 268 eine mögliche Realisierung dieser Schnittstellen für die Produkte der PROBAnD-Methode vorstellen. Im folgenden Abschnitt werden wir auf die Zeiteigenschaften der Systeme näher eingehen, da diese für mögliche Kopplungen zu berücksichtigen sind.

Zeitliches Verhalten

In allen von uns betrachteten reaktiven Systemen gibt es eine Größe, die für das Systemverhalten relevant ist: die *Zeit*. Für eine Heizungsregelung ist es z. B. wichtig zu wissen, ob es sich um einen Winter- oder Sommertag handelt. Eine Helligkeitssteuerung wird oft so realisiert, dass Licht nach einer gewissen Periode der Inaktivität ausgeschaltet wird. Für einen Regelkreis kann das Abtast- bzw. Regelintervall relevant sein. Diese Größe, auf die in Modellen z. B. über eine oder auch mehrere Systemuhren oder globale Zeitvariablen (siehe dazu z. B. [Gra03]) zugegriffen werden kann, wollen wir im Folgenden *Modellzeit* (t_{M}) nennen.

Abhängig von der technischen Realisierung eines Software-Systems bestehen unterschiedliche Abhängigkeiten zwischen der Zeit, wie sie aus der Sicht des Modells gesehen wird (Modellzeit), und der in der Realität des modellierten Systems tatsächlich ablaufenden Zeit (*Realzeit*). Da der Begriff „Realzeit" allerdings leicht mit dem Begriff der „Echtzeit" (engl. „real-time", siehe weiter unten) verwechselt werden kann, wollen wir diese Zeit hier metaphorisch *Wanduhrzeit* (t_{W}) nennen. Zwei Arten von Abhängigkeiten zwischen t_{M} und t_{W} lassen sich dabei identifizieren:

- $t_{\mathrm{M}} \sim t_{\mathrm{W}}$: Es besteht eine Verbindung (proportionale Abhängigkeit) zwischen der Modell- und der Wanduhrzeit. Der Regelfall ist, dass jedes im Modell beobachtete Verstreichen einer gewissen Zeitspanne dieselbe Zeit in der Realität benötig ($t_{\mathrm{M}} = t_{\mathrm{W}}$). Besitzen Modell- und Wanduhrzeit andere Zeitskalen, so kann eine solche Abhängigkeit über einen konstanten Skalierungsfaktor s zwi-

schen t_M und t_W beschrieben werden ($t_M = s \cdot t_W$). Ist $s > 1$, dann wird eine Verlangsamung der Modellzeit beschrieben. Ist $0 < s < 1$, dann wird eine Beschleunigung der Modellzeit ggü. der Wanduhrzeit ausgedrückt.

- $t_M \neq t_W$: Zwischen der Modell- und der Wanduhrzeit besteht keine erkennbare Verbindung. Man kann auch sagen, dass es sich bei t_M um eine „simulierte" Zeit handelt, d.h. auch wenn eine Zeiteinheit aus der Sicht des Modells vergangen ist, kann die tatsächlich verstrichene reale Zeit völlig davon abweichen. Insbesondere ist auch erlaubt, dass die Modellzeit „stehen bleibt", also Aktionen im System ablaufen, ohne dass die Zeit t_M fortschreitet (siehe [Pri03]).

Wichtig ist eine Unterscheidung der „Realisierung von Zeit" bei der Kopplung von reaktivem System und dessen Umgebung, da die Art der Kopplung von der Verbindung der Zeiten t_M und t_W abhängt. Tabelle 9-1 zeigt die Möglichkeiten einer solchen Kopplung.

Tabelle 9-1. Möglichkeiten der Kopplung von reaktivem System und Umgebung

Abhängigkeit der Modellzeit		**Möglichkeiten der Kopplung**
Reaktives System	**Umgebung**	
$t_M \sim t_W$	$t_M \sim t_W$	direkt, Verschmelzung (nur bei simulierter Umgebung)
$t_M \sim t_W$	$t_M \neq t_W$	nicht sinnvoll
$t_M \neq t_W$	$t_M \sim t_W$	
$t_M \neq t_W$	$t_M \neq t_W$	Synchronisation, Verschmelzung

Unter dem ersten Fall lässt sich die endgültige Realisierung des reaktiven Systems einordnen. Da in diesem Fall die Umgebung real existiert, fällt deren „Modellzeit" immer mit der Wanduhrzeit zusammen. Auch das reaktive System weist eine Verbindung von t_M und t_W auf, um die realen Zeitanforderungen erfüllen zu können (z.B. das Einschalten des Lichts nach einer vorgegebenen Zeit). Da beide Systeme dieselbe Zeitbasis aufweisen, lassen sich diese direkt über die Systemschnittstellen koppeln.

Handelt es sich bei der Umgebung um eine Simulation derselben, dann bietet sich neben dieser direkten Kopplung auch eine Verschmelzung der beiden Systeme an. Dies ist allerdings nur dann möglich, falls beide im selben Formalismus beschrieben wurden. Sinnvoller ist eine solche Verschmelzung, wenn kein Bezug zur Wanduhrzeit existiert (Fall 4), weil dann das Gesamtsystem wegen der gemeinsamen Modellzeit mit höchstmöglicher Geschwindigkeit laufen kann und dadurch eventuelle Wartezeiten entfallen („Time-Warp").

Bei einer Verschmelzung sollte man allerdings berücksichtigen, dass in dem erhaltenem *geschlossenen System*, keine externe Kommunikation zwischen Umgebung und reaktivem System stattfindet und damit auch keine eventuellen Verzögerungen auf Grund der technischen Realisierung der Sensoren und Aktuatoren (siehe dazu [Que02, S.181f.]) oder des Kommunikationsmediums beobachtet werden können. Ist dies für eine Überprüfung des Gesamtsystems relevant, so muss man entweder auf eine solche Kopplung verzichten oder aber die Eigenschaften der Kommunikationsschicht und der Sensoren und Aktuatoren in das Umgebungsmodell aufnehmen (siehe [Rie04]).

Ist im Fall 4 keine Verschmelzung des reaktiven Systems mit seiner Umgebung (Simulation) möglich, dann wird eine Synchronisation nötig, weil anders als in Fall 1 von keiner gemeinsamen Zeit ausgegangen werden kann. Im Detail bedeutet dies, dass erst dann, wenn eines der Systeme seine Ergebnisse für einen Zeitpunkt berechnet hat, das jeweils andere System auf der Basis dieser Ergebnisse weiterrechnen darf. Da man den Zeitpunkt des Ergebnisses einer Berechnung in der Zeit t_{W} nicht kennt, kann man nicht einfach „warten“, sondern muss eine Synchronisation z. B. über ein Handshake-Mechanismus realisieren. Hiermit wird – wie bei einer Verschmelzung – eine maximale Berechnungsgeschwindigkeit erreicht.

Auch im Fall 1 kann man bei Existenz einer simulierten Umgebung eine Beschleunigung der Berechnungen erreichen. Hierzu muss man nur eine gleichwertige Reduzierung des Skalierungsfaktors s in beiden Systemen vornehmen. Jedoch erreicht man damit i. d. R. keine maximale Beschleunigung, da s so gewählt werden muss, dass auch der langsamste Rechenschritt noch „pünktlich“ abgeschlossen wird. In Gebäudesimulatoren konnte Zimmermann mit diesem Ansatz z. B. eine Beschleunigung bis zu einem Faktor von 100 erreichen ($s = 1/100$, vgl. [Zim02]).

Auf eine mögliche technische Umsetzung obiger Arten der Kopplung für die Fälle 1 und 4 werden wir in Abschnitt 9.2.3 auf Seite 268 eingehen. In den Fällen 2 und 3 ist eine Kopplung von reaktivem System und Umwelt – wenn auch technisch durch die Kopplung der Schnittstellen realisierbar – nicht sinnvoll.

Da im Fall 2 die Umgebung (Simulation) losgelöst von der Zeit t_{W} ist, kann die reale Zeit, die während einer Einheit der Modellzeit t_{M} verstreicht nicht nur geringer sondern auch länger als die Modellzeit sein (z. B. bei komplexen numerischen Berechnungen). In einer solchen Situation wird das beobachtete Verhalten des reaktiven Systems wegen der verspäteten Antwort der Umgebung von dem erwarteten Verhalten abweichen. Inwieweit ein solcher Testlauf dann für eine vernünftige Verifikation oder Validierung eingesetzt werden kann ist fraglich, da man das reaktive System quasi in einer anderen Umgebung getestet hat.

Im Fall 3 verbieten die Abschnitt 6.1.1 auf Seite 132 formulierten Anforderungen an ein reaktives System eine Kopplung. Denn auch wenn eventuelle Zeitanforderungen, die im Modell formuliert sind, in der Modellzeit erfüllt werden, kann daraus

nicht gefolgert werden, dass die Reaktionen auf Stimuli schnell genug in der Umgebung ankommen, da das reaktive System hier keinen Bezug zur „Wanduhr“ aufweist.

Im Zusammenhang mit den obigen Zeitbegriffen sei zum Abschluss kurz auf den Begriff der „Echtzeit“ eingegangen. Die Forderung nach einer Echtzeitverarbeitung bedeutet allgemein, dass die an das System gestellten Zeitanforderungen eingehalten werden, also Ergebnisse innerhalb definierter Zeitspannen geliefert werden (siehe z.B. [Dou99, S.7ff.]). Für ein reaktives System entspricht dies der in Abschnitt 6.1.1 auf Seite 132 aufgestellten Forderung nach „Rechtzeitigkeit“. Forderte man z.B. für eine Helligkeitssteuerung, dass spätestens 300ms nachdem man einen Raum betreten hat, die gewünschte Helligkeit erreicht wird, wäre eine solche Steuerung in den Augen des Benutzers wertlos, wenn das Licht erst nach 2 Minuten eingeschaltet wird.

In der Literatur wird dieser Echtzeitbegriff weiter in die sog. *harte*, *weiche* und *feste Echtzeit* unterteilt [Nic98, S.7]. Bei der harten Echtzeit bedeutet ein zu spätes Ergebnis immer ein fehlerhaftes Systemverhalten, das zu einer ökonomischen oder menschlichen Gefährdung führen kann (z.B. bei einer Notabschaltung eines Kernkraftwerks). Nimmt die Nützlichkeit des Ergebnisses mit der Größe der Zeitverletzung ab [BrA03, S.5], dann spricht man von weicher Echtzeit. Einige der Zeitspannen dürfen manchmal überschritten werden, ohne dass die Systemfunktionalität leidet. Im obigen Beispiel wäre es sicherlich auch akzeptabel (wenn auch nicht ideal), wenn das Licht erst 600ms nach Betreten des Raumes eingeschaltet wird. Auch wenn das Licht in sehr seltenen Fällen mal eine Sekunde bis zum Einschalten braucht, führt dies nicht direkt zu einer Gefährdung. Zwischen der weichen und der harten Echtzeit ist die feste Echtzeit anzusiedeln. Es dürfen wie bei der weichen Echtzeit Zeitanforderungen verletzt werden, nur ist in einem solchen Fall das Ergebnis wertlos.

Wie wir an dem Beispiel des Lichts bereits illustrierten, sind in der Domäne der Gebäudeautomationssysteme zu einem sehr großen Anteil nur weiche Echtzeitanforderungen relevant. Wir werden daher – wie auch Queins in seiner Arbeit zur PROBAnD-Methode – in unseren Automatisierungsbeispielen nur diese Art nichtfunktionaler Anforderungen betrachten. Konkret bedeutet dies, dass wir in der Anforderungsspezifikation nur weiche Zeitanforderungen formulieren. Diese können dann z.B. in einem Prototyping-Schritt validiert und an die anschließende Design-Phase weitergereicht werden (siehe dazu auch [Que02, S.216ff.]).

9.1.4 Existierende Ansätze zur Prototypgenerierung

Prototyping wird mehr oder weniger seit den 1980er Jahren in der Software-Entwicklung eingesetzt. Dementsprechend viele Veröffentlichungen und Arbeiten sind

auf diesem Gebiet zu finden. Neben dem reinen Software-Prototyping wird auch häufig Prototyping im Kontext des Hardware-Software-Co-Designs betrachtet. Die Proceedings des jährlich stattfindenden IEEE Workshops „Rapid System Prototyping“ beinhalten zu dieser Thematik sehr gute Beiträge (siehe z.B. [KoA03]).

In dieser Arbeit wollen wir uns auf das Software-Prototyping beschränken und – wie bereits in Abschnitt 9.1.3 angedeutet – das Prototyping von grafischen Benutzerschnittstellen nicht betrachten, da sich unsere Arbeiten auf die Entwicklung reaktiver Systeme konzentrieren. Im Folgenden werden wir einen Überblick über dieses Gebiet an Hand einiger wichtiger Ansätze zur automatischen Prototyperstellung aus einer Prototypspezifikation geben.

In der Literatur findet man zur Prototypgenerierung zwei prinzipielle Ansätze. Bei dem ersten Ansatz wird ein möglichst abstrakter Formalismus zum Einsatz gebracht, der von konkreten Design- oder Implementierungsentscheidungen abstrahiert. Beim zweiten Ansatz werden aus (unvollständig) beschriebenen Szenarien ablauffähige Prototypen synthetisiert.

Abstrakte Formalismen

Als abstrakte Formalismen eignen sich die sog. Very-High-Level-Languages (VHLL, siehe [BKK92, S.131ff.]), zu denen z.B. Prolog als prädikative Programmiersprache zählt. Heutzutage nimmt allerdings die Bedeutung von Modellierungssprachen zu (die in der Regel eine grafische Syntax besitzen). Beide Arten von Formalismen sind für die Beschreibung jeweils unterschiedlicher Klassen zu selektierender Eigenschaften sinnvoll.

Im Folgenden wird eine Auswahl existierende Formalismen und deren mögliche Anwendungsfelder aufgezeigt:

- *Petri-Netze* [Rei90]: Viele Petri-Netz-Varianten wurden bereits für das Prototyping eingesetzt. Eine Übersicht findet sich z.B. in der Arbeit von Kordon [Kor94]. Beispielhaft sei hier der Ansatz von Ebert erwähnt, in welcher „Coloured Predicate/Transition Nets“, die hierarchisch verfeinert werden können, für die Entwicklung reaktiver Systeme eingesetzt werden [Ebe98]. Der grundsätzliche Vorteil der Petri-Netze liegt in der Abstraktion von der tatsächlichen Abarbeitungsreihenfolge der Aktionen und der äußerst einfachen Notation, die auch Laien zugänglich gemacht werden kann.
- *MATLAB/Simulink* [Hof98]: Sehr gut geeignet zur Beschreibung und Ausführung kontinuierlicher Systeme sind die Blockdiagramme von MATLAB/Simulink, wobei auch die Kombination mit einer zustandsorientierten Modellierung möglich ist (Simulink/Stateflow).

- *Zustandsautomaten*: Diese wurden bereits in Abschnitt 6.2.3 auf Seite 151 ausführlich behandelt und deren Vorzüge für die Domäne der reaktiven Systeme herausgestellt.

Für die Realisierung eines frühen Prototypings in der PROBAnD-Methode wurden die obigen Ansätze untersucht und alle außer den Zustandsautomaten wieder verworfen, da ein zu starker Bruch in der Systementwicklung auftrat. Da sich die Semantik von z. B. Petri-Netzen sehr stark von der Semantik der Zustandsautomaten unterscheidet, konnte damit keine wirklich durchgängige Entwicklung realisiert werden. Neben dem Aufwand des „Erlernens“ von zwei unterschiedlichen Sprachen sahen wir auch die Schwierigkeit in der jeweiligen Begriffswelt sauber zu modellieren und z. B. nicht in Versuchung zu geraten, mit Hilfe von Petri-Netzen eine stark zustandsorientierte Beschreibung vornehmen zu wollen.

Desweiteren gelang es uns bei keinem dieser Ansätze eine größere Spezifikation so in Teile zu zerlegen, dass eine vernünftige Reduktion der Komplexität erreicht wurde. Erst mit unserer modifizierten Automatenmodellierung aus Abschnitt 6.2.3 auf Seite 151 glückte uns dieser Schritt.

Synthese aus Szenarien

Die abstrakten Formalismen abstrahieren von dem tatsächlichen Verhalten des Zielsystems. Bei den szenario-basierten Ansätzen wird hingegen ein anderer Weg begangen. Ausgehend von gegebenen Szenarien werden ablauffähige Prototypen (oder in einem ersten Schritt Prototypspezifikationen) synthetisiert. Eine allgemeine Beschreibung dieses Ansatzes wird von Harel in [Har01] präsentiert.

Unter Szenario versteht man die „*partielle* Beschreibung der Systemverwendung aus der Sicht der Benutzer oder angrenzender Systeme“ [AmE03]. Amyot und Eberlein stellen einen sehr umfangreichen Vergleich existierender Szenario-Notationen und der darauf aufsetzenden Techniken zur Synthese von Design-Modellen vor, aus welchen in vielen Fällen Prototypen generiert werden können [AmE03].

Beispielhaft seien hier drei dieser Ansätze aufgeführt:

- Whittle und Schumann stellen in [WiS00] einen Ansatz für die Synthese von UML Statecharts aus UML Sequenzdiagrammen vor.
- Leue et al. schlagen in [LMR98] die Generierung von ROOM-Modellen (Real-Time Object-Oriented Modeling [Wie98, S. 517f.]) aus HMSC-Szenarien (High-Level MSC [MaR97]) vor.
- Mansurov und Zhukov präsentieren ein Verfahren zur Erzeugung von SDL-Modellen aus HMSCs [MaZ99]. Die kommerzielle Version ihrer Arbeiten ist als „KLOCwork MSC to SDL Synthesizer“ mittlerweile Teil der Telelogic Tau SDL Suite.

Als wichtigste Nachteile dieser Ansätze identifizieren Amyot et al. die oftmals notwendige Einschränkung der Ausgangs- und Zielnotationen, um eine vernünftige Automatisierung zu realisieren. Desweiteren kritisieren sie die Tatsache, dass die meisten synthetisierten Modelle kaum verständlich und dadurch wenig wart- und erweiterbar sind. Ich bin der Meinung, dass diese Unverständlichkeit zum Teil darauf zurückgeführt werden kann, dass die Zustände der typischerweise synthetisierten Zustandsautomaten keine wirklich vernünftige Bedeutung tragen, da diese oftmals nicht den beobachtbaren Zuständen der Realweltobjekte entsprechen.

Wie bei der Definition des Szenariobegriffs schon hervorgehoben wurde, stellen Szenarien immer nur exemplarische Abläufe eines Systems dar. Auch wenn durch mächtige Konzepte wie Wiederholungen, parallele Abschnitte und Übersichtsszenarien (wie z.B. in HMSCs) viele solcher Abläufe gleichzeitig in einem Diagramm beschrieben werden können (siehe z.B. [JRH03, S.329ff.]), wird es bei komplexen System nicht praktikabel oder möglich sein, das vollständige Verhalten mit Szenarien zu beschreiben. Insbesondere, da reaktive Systeme auf Stimuli, die zu jedem beliebigen Zeitpunkt auftreten können, reagieren müssen, hege ich daher Zweifel, inwiefern auf der Basis unvollständiger Information ein Prototyp mit einer realistischen Funktionsweise erzeugt werden kann. In anderen Domänen scheint sich allerdings die Spezifikation von Szenarien vernünftig in einen Entwicklungsprozess integrieren zu lassen. Allerdings ist auch hier die Frage nach Vollständigkeit und Konsistenz von Szenarien nicht einfach zu beantworten, wie auch Amyot et al. in [AmE03] herausstellen.

In meinen Augen besitzen alle szenario-basierten Ansätze desweiteren das Manko, nicht für wirklich komplexe Systeme geeignet zu sein. Da ein Szenario immer die Interaktion von Instanzen zeigt, skaliert ein solches Modell bei einer zunehmenden Zahl von Objekten nicht vernünftig (bei unserer Gebäudeautomationsfallstudie *FloorAutomationX* aus Abschnitt 8.2.2 auf Seite 224 wurden ca. 1000 Instanzen bei der Spezifikation des Systems berücksichtigt). Reduziert man aus diesem Grund den durch ein Szenario dargestellten Ausschnitt, dann werden die Modelle i.d.R. trivial und tragen nichts zum Verständnis oder zur Komplexitätsbeherrschung bei. Konsequenterweise sind die obigen Syntheseansätze nicht vernünftig auf unsere Klasse von Systemen anwendbar.

Dienen die aus Szenarien erzeugten Prototypen lediglich der Animation genau definierter Abläufe, dann kann diese Technik sinnvoll sein. Auch der Einsatz von Szenarien als Beschreibung von Testfällen ist in vielen Fällen angebracht und wird auch von vielen Werkzeugen (z.B. von Telelogic Tau und iLogix Rhapsody) unterstützt. Interessant ist hierbei vor allem der Vergleich der Szenarien zur Testfallbeschreibung mit den tatsächlich zur Laufzeit des Prototyps aufgezeichneten Szenarien (siehe dazu z.B. [Dol03, S.111ff.], [Har01] und Abschnitt 9.3.1 auf Seite 276).

Auch wenn Szenarien in der obigen Form für unsere Entwicklungsmethode nicht geeignet sind, muss man den Beitrag der Szenario-Notation zum Schließen der Lücke zwischen der Problembeschreibung und einer operationalen Anforderungsspezifikation (z.B. mit Zustandsautomaten) anerkennen. Insbesondere da Untersuchungen bzgl. menschlicher Problemlösestragien zeigten, dass es einfacher ist typische Einsatzszenarien für ein hypothetisches System zu formulieren als die Eigenschaften eines solchen Systems zu spezifizieren [AmE03]. Wir begannen daher mit der Entwicklung einer grafischen Notation, welche diese Lücke in der PROBAnD-Methode zu schließen vermag. Im Ausblick dieser Arbeit (Abschnitt 11.4.2 auf Seite 353) werden wir nochmals darauf eingehen.

9.2 Automatische Prototyperstellung

Nachdem wir im vorangegangenen Abschnitt die Grundlagen des Prototypings geklärt haben, wollen wir nun zeigen, wie auf der Basis der im ersten Teil dieser Arbeit vorgestellten Automatisierungstechnik die Erstellung von Prototypen automatisiert werden kann.

Eine automatische Prototyperstellung wird dabei in unserem Ansatz an zwei Stellen unterstützt:

1. *Eigenschaftsselektion*: Da die Anforderungsspezifikation als eine Instanz des PROBAnD-Produktmodells vorliegt (siehe Abschnitt 6.2 auf Seite 148) und damit in einer formalen Syntax beschrieben ist, kann eine Selektion der relevanten Eigenschaften durch eine Transformation dieser Produktmodellinstanz beschrieben werden.
2. *Prototypkonstruktion*: Durch einen Unparse-Schritt lässt sich aus der unter Punkt 1 erzeugten Produktmodellinstanz eine Prototypspezifikation in SDL erzeugen, aus welcher ausführbare Prototypen mit dem kommerziellen Werkzeug Telelogic Tau erzeugt werden können.

Einen Teil des zweiten Schritts hatten wir bereits in Abschnitt 7.2.1 auf Seite 181 erläutert. Abbildung 9-4 zeigt die Schritte und die eingesetzten Werkzeuge auf dem Weg von der Anforderungsspezifikation zum ausführbaren Prototypen grafisch.

In den folgenden Unterabschnitten werden wir genauer auf Punkt 1 (die Eigenschaftsselektion) eingehen.

9.2.1 Horizontales Prototyping

Um ein horizontales Prototyping zu unterstützen, muss man eine Eigenschaftsselektion automatisieren, welche zu einer Reduktion des Detaillierungsgrads der Implementierung führt (siehe Abschnitt 9.1.1 auf Seite 242). Eine solche Selektion lässt

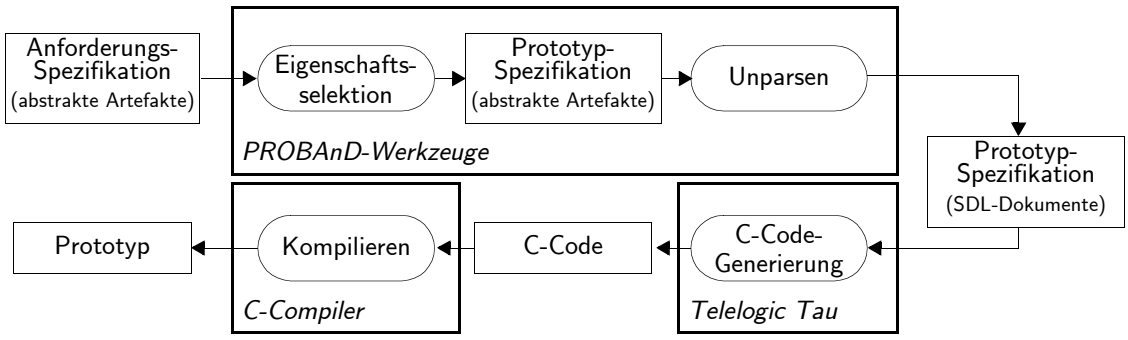


Abbildung 9-4. Automatische Prototyperstellung und Werkzeugeinsatz

sich im Kontext der PROBAnD-Methode sehr leicht durch eine strukturelle Ausschnittsbildung realisieren.

Strukturelle Ausschnittsbildung

Bei einer strukturellen Ausschnittsbildung werden diejenigen Control-Object-Types eliminiert oder durch entsprechende „Platzhalter“ ersetzt, von deren Funktionalität abstrahiert werden soll.

Eine solche Elimination ist in vielen Situationen möglich, da in einer PROBAnD-Spezifikation die aggregierten Control-Object-Types in aller Regel zur Realisierung der Funktionalität des aggregierenden Objekttyps beitragen. Daher kann man eine Eigenschaftsselektion beim horizontalen Prototyping unterstützen, indem man diejenigen Control-Object-Types, die sich weiter unten im Aggregationsbaum befinden, durch *Stubs* [May90, S.449f.] ersetzt. Diese Stubs „simulieren“ die eliminierten Komponenten mit einer gewünschten Genauigkeit. Abb. 9-5 zeigt wie eine solche Abstraktion prinzipiell aussehen kann.

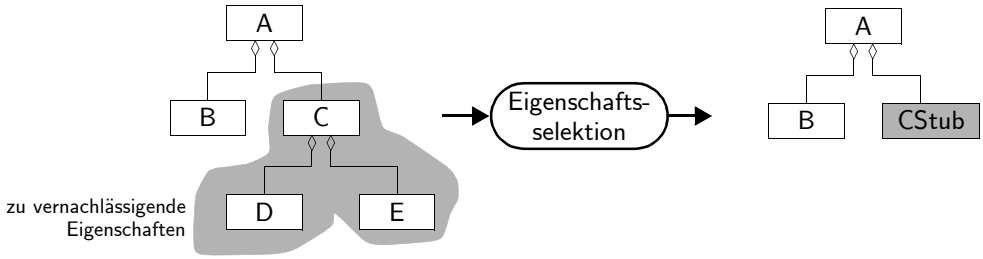


Abbildung 9-5. Strukturelle Abstraktion durch Einsatz eines Stubs

Zur Veranschaulichung einer strukturellen Abstraktion könnte man in unserem *RoomAutomation*-Beispiel aus Abschnitt 6.1.3 auf Seite 138 z.B. den Temperatursensor TempSens durch einen einfachen TempSensStub ersetzen, der stets eine konstante Temperatur liefert.

Als mögliche Typen von Stubs, die automatisch generiert werden können, wurden exemplarisch die folgenden Möglichkeiten realisiert:

1. *Struktureller Stub*: Ein Stub eines solchen Typs beinhaltet keinerlei Verhalten und dient lediglich als „Platzhalter“ in der Systemstruktur. Insbesondere bedeutet dies, dass die Instanzen dieses Stubs existieren und daher in den Verhaltenskonstrukten der Control-Object-Types, welche diese Instanzen aggregieren, darauf Bezug genommen werden kann (siehe sentTo-Relation im Produktmodell aus Abb. 6-14 auf Seite 157).

2. *Passiver Stub*: In den SDL-Modellen ist neben der Existenz der Zielinstanz einer Sendeaktion (welche einen entsprechenden Kanal impliziert, siehe Abb. 6-6 auf Seite 145) auch die Existenz eines korrespondierenden Konstrukts für den Signalempfang notwendig. Ansonsten ginge ein gesendetes Signal (Signalinstanz) verloren (vgl. dazu auch Szenario a) in Abb. 9-3). Ein passiver Stub wird daher so realisiert, dass für jeden in dem ursprünglichen Objekttyp konsumierten Signaltyp (identifiziert über die consumes-Relation zwischen dem abstrakten Artefakt Strategy und dem Artefakt SignalType, siehe Abb. 6-10 auf Seite 150) eine Transition mit einem entsprechenden ReceiveEvent eingeführt wird. Damit ist gewährleistet, dass der Stub nicht die Ursache für verlorene Signale sein kann. Dies vereinfacht die Suche solcher Signalinstanzen in anderen Teilen des Systems.

3. *Randomisierter Stub*: Als eine Erweiterung des passiven Stubs ist der randomisierte Stub zu betrachten, der neben dem Empfang von Instanzen der konsumierten Signaltypen auch ein aktives Verhalten aufweist. Ein randomisierter Stub wird dabei so realisiert, dass für jeden Signaltyp, der von einem aggregierenden Objekttyp empfangen werden kann, Signalinstanzen zu zufälligen Zeitpunkten und mit zufälligen Parameterbelegungen versendet werden. Die gewählte Verteilung der Zufallsvariablen für Sendezeitpunkt und Parameterbelegung ist die Gleichverteilung, wobei für jede Zufallsvariable eine untere und obere Grenze für den Wertebereich angegeben werden kann.

Will man Stubs erzeugen, die über mehr Funktionalität als die obigen Varianten verfügen (z.B. die erwähnte Erzeugung der stets gleichen Temperatur oder komplexere Approximationen des Verhaltens) gibt es zwei Möglichkeiten: Entweder man nutzt unsere Technik zur Automatisierung und generiert das gewünschte Verhalten durch eine Modelltransformation oder man erzeugt Stubs nach dem Verfahren in Punkt 1 (also ohne Verhalten) und beschreibt das gewünschte Verhalten manuell. Dazu kann man z.B. aus den abstrakten Artefakten der Prototypspezifikation neben der SDL-Dokumente auch die HTML-Dokumente der PROBAnD-Methode erzeugen und diese modifizieren.

Im Folgenden wollen wir zeigen, wie wir die obige Generierung der verschiedenen Arten von Stubs mit Hilfe unserer Multiebenenaktionssprache AL++ implementieren können. Dabei werden wir zur Wahrung der Lesbarkeit die Modifikationen der

komplexen Artefakte, welche nur bzgl. der aggregierten atomaren Artefakte verändert werden, auslassen.

Beginnen wollen wir mit dem einfachsten Fall, dem strukturellen Stub, welcher durch die Methode `convertToStructuralStub()` des Artefakttyps ControlObjectType erzeugt wird:

```
1   /* ControlObjectType: */
2   public void convertToStructuralStub() {
3       name = name+"Stub";
4       ControlObjectType childCot;
5       foreach(Instantiation inst in aggregatedInstantiation) {
6           childCot = inst.instantiatedControlObjectType;
7           childCot.instantiatingInstantiation -= inst;
8       }
9       aggregatedInstantiation -= all;
```

Zunächst wird an den Namen des Control-Object-Types der String „Stub“ angehängt, um dessen modifizierten Charakter zu kennzeichnen. Vollständigerweise müsste man hier noch auf einen eventuellen Namenskonflikt prüfen und dann einen anderen Namen vergeben, indem man z.B. solange Zahlen an den Namen anhängt, bis dieser kollisionsfrei ist (weitere Möglichkeiten werden von Kohlbecker et al. in [KFF86] diskutiert).

Danach wird der Teilbaum unterhalb des Stubs entfernt. Dazu läuft man über alle Instanziierungen, welche vom ursprünglichen Objekttyp aggregiert wurden. In jedem durch eine solche Instanziierung instanziierten Control-Object-Type entfernt man dann den instantiates-Link (siehe auch Abb. 6-9 auf Seite 149). Ist dies für alle aggregierten Objekttypen erfolgt, kann man auch alle aggregates-Links löschen (Zeile 7).

Nach der Anpassung der Struktur muss nun als Nächstes das „alte“ Verhalten des Objekttyps gelöscht werden:

```
10      Strategy strat;
11      foreach(Task task in implementedTask) {
12          strat = task.realizingStrategy;
13          strat.realizingTransition -= all;
14          strat.readAttribute -= all;
15          strat.writtenAttribute -= all;
16          strat.modifiedTimer -= all;
17          strat.receivedTimer -= all;
18      }
19  }
```

Dies ist recht einfach möglich, indem man für alle von dem Objekttyp implementierten Tasks alle Links zu den realisierten Transitionen und den von diesen Tasks verwendeten Attributen und Timern entfernt. Auch die Links zu den Signaltypen könnten so entfernt werden. Da wir allerdings von der durch diese Links beschriebenen Verfolgbarkeitsrelation bei der Erzeugung der passiven Stubs Gebrauch machen werden, führen wir eine solche Löschung nicht durch.

Die Erzeugung eines passiven Stubs erfolgt durch die nun folgende Operation `convertToPassiveStub()`, wobei zunächst durch den Aufruf der Operation `createStructuralStub()` ein „verhaltensleerer" Stub generiert wird.

```
1   /* ControlObjectType: */
2   public void convertToPassiveStub() {
3       convertToStructuralStub();
4
5       Set consumedSigTypes = new HashSet();
6       SignalType sigtype;
7       Strategy strat;
8       foreach(Task task in implementedTask) {
9           strat = task.realizingStrategy;
10          consumedSigTypes.addAll(strat.consumedSignalType);
11      }
12
13      Set producedSigTypes = new HashSet();
14      ControlObjectType parentCot;
15      foreach(Instantiation inst in instantiatingInstantiation) {
16          parentCot = inst.aggregatingControlObjectType;
17          foreach(Task task in parentCot.implementedTask) {
18              strat = task.realizingStrategy;
19                  producedSigTypes.addAll(strat.producedSignalType);
20          }
21      }
22      consumedSigTypes.retainAll(producedSigTypes);
```

Nicht alle vom betrachteten Objekttyp konsumierten Signaltypen dürfen berücksichtigt werden. Viele der konsumierten Signale kamen nämlich ursprünglich von den aggregierten Control-Object-Types. Würde man auch für solche Signale eine Transition mit einem entsprechenden ReceiveEvent erzeugen, würde ein fehlerhaftes SDL-Modell (und damit eine fehlerhafte Prototypspezifikation) erzeugt, da es keinen Kanal (SDL-Channel) gäbe, auf welchem ein solches Signal laufen könnte.

Aus diesem Grund müssen wir zunächst die Menge derjenigen Signaltypen feststellen, welche von den aggregierenden Objekttypen (Elternknoten im Baum) ausgehen. Dazu berechnet man die Menge der konsumierten Signaltypen (Zeilen 5 bis 11) und die Menge der von allen Elternknoten (parentCot) produzierten Signaltypen

(Zeilen 13 bis 20). Die Schnittmenge (berechnet über die Java-Methode `retainAll()`) beinhaltet dann die gewünschten Signaltypen.

Ist diese Menge nichtleer, dann wird wie folgt verfahren:

```
23      String taskName = this.name+"T1";
24      Task task = new Task(taskName);
25      this.implementedTask += task;
26      strat = new Strategy(taskName+"Strategy");
27      task.realizingStrategy = strat;
28      State idleState = new State("idle");
```

Um die im Folgenden erzeugten Transitionen einem Task zuordnen zu können (Verfolgbarkeit) wird ein solcher neu erzeugt (auch hier müsste man wie oben auf Namenskollisionen prüfen). Neben diesem Task wird auch eine entsprechende Strategy instanziiert und mit diesem Task verbunden. Zuletzt generiert man noch eine Instanz des Artefakttyps State, da der Zustand idle Ausgangs- und Zielzustand der Transitionen wird.

```
29      Transition trans;
30      foreach(sigtype in consumedSigTypes) {
31          trans = new Transition("");
32          strat.realizingTransition += trans;
33          StateSet sts = new StateSet("");
34          sts.includedState += idleState;
35          trans.originatingAbstractStateSet = sts;
36          trans.terminatingState = idleState;
37
38          ReceiveEvent event = new ReceiveEvent("");
39          Signal sig = new Signal(sigtype.name);
40          sig.itsSignalType = sigtype;
41          event.creatingSignal = sig;
42          trans.triggeringEvent = event;
43
44          ActualParameter aparam;
45          String paramDescr;
46          foreach(FormalParameter fparam in sigtype.itsFormalParameter) {
47              paramDescr = "pr"+sigtype.name+fparam.parameterNbr;
48              aparam = new ActualParameter("");
49              aparam.description = paramDescr;
50              aparam.parameterNbr = fparam.parameterNbr;
51              sig.itsActualParameter += aparam;
52
53              Attribute attr = new Attribute(paramDescr);
54              attr.itsDataType = fparam.itsDataType;
```

```
55                  strat.writtenAttribute += attr;
56              }
57          }
58      }
```

Für jeden Signaltyp, den es zu betrachten gilt, wird in Zeile 29–36 eine neue Transition erzeugt, die – wie oben schon erwähnt – den Zustand idle sowohl als Ausgangszustand (originatesAt-Relation aus Abb. 6-12 auf Seite 154) und als Zielzustand besitzt (terminatesAt-Relation).

Dann wird in den Zeilen 38 bis 42 für den aktuell betrachteten Signaltyp ein ReceiveEvent erzeugt, das über die creates-Assoziation mit einer entsprechenden Instanz des Artefakttyps Signal verbunden wird. Diese Instanz besitzt denselben Namen wie der zu konsumierende Signaltyp und referenziert diesen über die isOf-Relation (siehe Abb. 6-14 auf Seite 157).

Als Nächstes müssen die aktuellen Parameter von Signal definiert werden. Dazu wird für jeden formalen Parameter des Signaltyps eine entsprechende Instanz des Artefakttyps ActualParameter erzeugt. Da die aktuellen Parameter in derselben Reihenfolge wie die formalen Parameter im Signal auftauchen müssen, wird das Attribut parameterNbr entsprechend identisch belegt. Der Inhalt (description) der Parameter entspricht dabei dem Namen des Attributs, in welchem der empfangene Wert des Signalparameters gespeichert werden soll.

Diese Attribute werden nun als Letztes definiert. Dazu genügt es eine entsprechende Attributinstanz mit demselben Namen wie die paramDescr anzulegen und dieses Attribut als von der Strategie geschriebenes Attribut kenntlich zu machen.

Die Erzeugung eines randomisierten Stubs erfolgt in analoger Weise zu dem passiven Stub, nur dass hierbei konsumierte/produzierte Signale und Aktionen/Ereignisse die Rollen tauschen. An Stelle des Codes zur Generierung solcher Stubs wollen wir hier zeigen, wie sich der Einsatz eines randomisierten Stubs auf unser *RoomAutomation*-Beispiel auswirkt.

Angenommen, wir hätten noch keine Umgebung (oder eine Simulation derselben), um unser System zu testen, dann böte der Einsatz randomisierter Stubs die Möglichkeit, Zufallswerte für Sensoren zu erzeugen (siehe auch Abschnitt 9.3.1 auf Seite 276). In Abb. 9-6 ist gezeigt, wie mit Hilfe des Werkzeuges *Clipper* ein randomisierter Stub für den Temperatursensor (TempSens) erzeugt wird.

Für jeden Signaltyp wird dabei eine zeitliche Verteilung dessen Signalinstanzen (hier von 10 bis 100) und die Verteilung der Werte der Signalparameter (hier von 18 bis 22) abgefragt. Die generierten Transitionen des TempSensStubs sind in Abb. 9-7 gezeigt.

Wie man erkennt, wird neben der Funktionalität zum Empfang von Signalen (Transition für den Task TempSensStubT1) auch die entsprechenden Transitionen

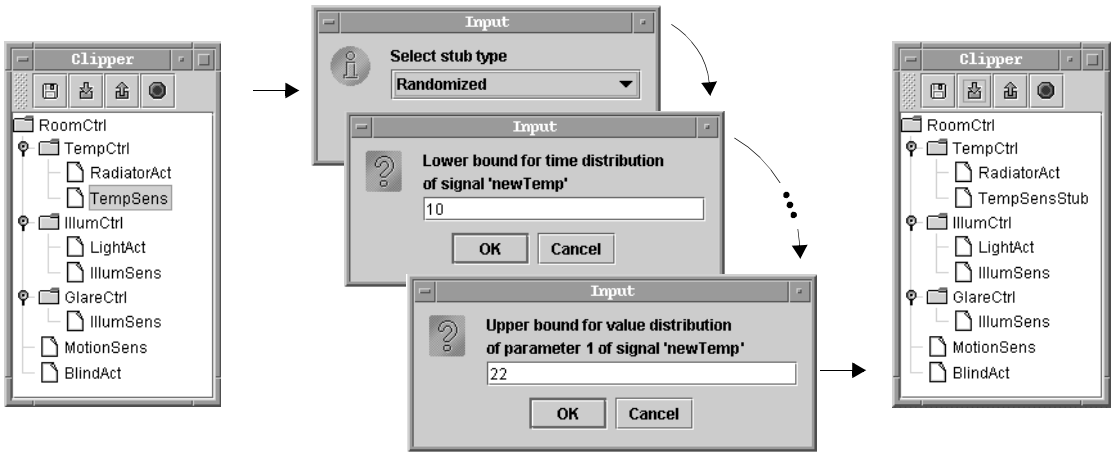


Abbildung 9-6. Bildschirmabzug einer Sitzung zur strukturellen Abstraktion

zur Realisierung der zufälligen Signalerzeugung generiert (Transitionen, die mit dem Task TempSensStubT2 markiert sind).

Für jede zufällige Größe (im Beispiel für den Timer tNewTemp und den Parameter psNewTemp1) wird dabei ein Pseudo-Zufallsgenerator (rctNewTemp und rcpsNewTemp1) definiert. Der zur Definition dieser Generatoren verwendete Datentyp RandomControl ist eine werkzeugspezifische Erweiterung von Telelogic Tau und ist in einer entsprechende „Random“-Bibliothek definiert (siehe dazu [Tel03]).

Zur Initialisierung werden diese Zufallsgeneratoren mit dem Startwert 1 („Random Seed“) initialisiert, um wiederholbare Ergebnisse zu liefern. (Will man mehr „Zufälligkeit“, so kann man diesen Startwert ebenfalls zufällig auswählen.)

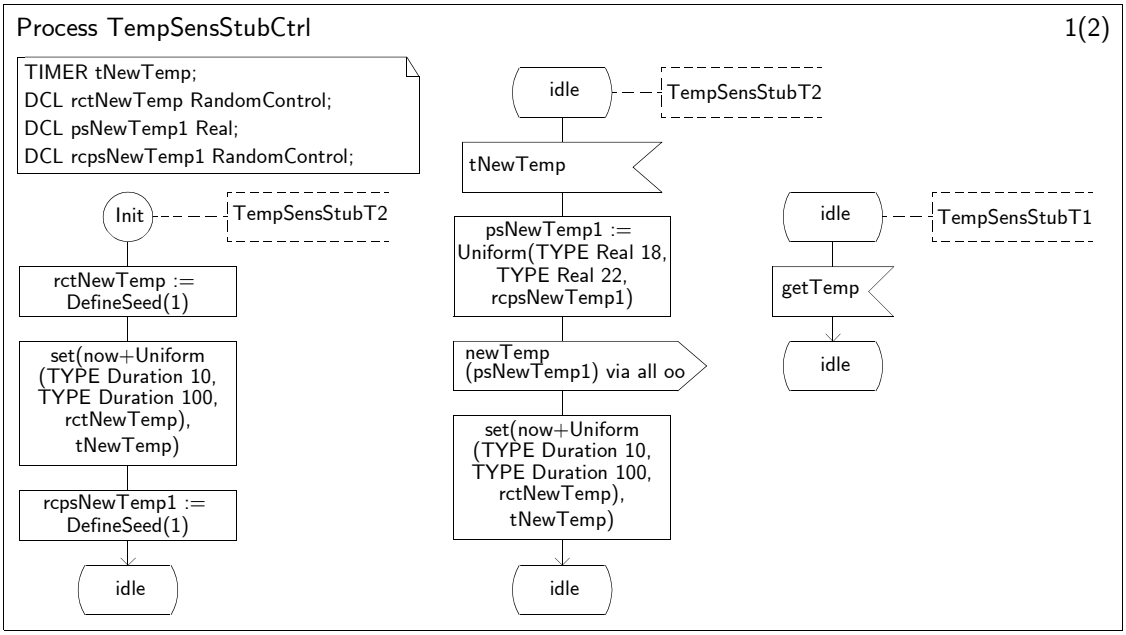


Abbildung 9-7. Randomisierter Stub in SDL-Notation

Danach wird der Timer zur Signalinstanzerzeugung mit einer zufälligen Dauer aus dem Intervall 10 bis 100 aufgezogen. Führt dieser Timer zu einem Timer-Ereignis, wird ein zufälliger Wert aus dem festgelegten Intervall berechnet. Dieser wird als aktueller Parameter des Signals newTemp verwendet, welches an die aggregierende Instanz gesendet wird. Abschließend wird der Timer wieder auf einen zufälligen Zeitpunkt aufgezogen.

Eine solche Berechnung von zufälligen Werten macht allerdings nur für zahlenwertige Parameter (also für Integer, Duration, Real u.ä. Datentypen) Sinn. Bei Parametern vom Typ String oder komplexeren Datentypen wird daher auf zufällig generierte Werte verzichtet und die Eingabe einer Konstante im Werkzeug verlangt.

Verhaltensabstraktion

Neben einer strukturellen Ausschnittsbildung ist auch eine Verhaltensabstraktion bei der Eigenschaftsselektion möglich. In der PROBAnD-Methode wird eine solche Form der Abstraktion dadurch ermöglicht, dass die Transitionen, welche eine Strategie beschreiben, nicht in ihrem endgültigen Detaillierungsgrad formuliert werden müssen. So kann insbesondere die Realisierung der Prozeduren, welche während einer Transition ausgeführt werden, ausgelassen oder vereinfacht dargestellt werden.

Eine automatische Unterstützung dieser Art der Abstraktion wurde in dieser Arbeit allerdings nicht untersucht. Prinzipiell wäre dies aber ebenso auf der Basis der detaillierten Verfolgbarkeitsrelationen, die bereits im vorhergehenden Abschnitt zur Implementierung des Automatisierungsalgorithmus eingesetzt wurde, denkbar.

9.2.2 Diagonales Prototyping

Da wir in der PROBAnD-Methode die Prototypgenerierung nur ausgehend von SDL-Modellen unterstützen, ist ein echtes vertikales Prototyping, bei dem ein Teil des Systems in seiner endgültigen Form implementiert wird, nicht möglich. Allerdings lässt sich natürlich eine Ausschnittsbildung aus der Gesamtspezifikation automatisieren und damit ein diagonales Prototyping (siehe Abschnitt 9.1.1 auf Seite 242) realisieren.

Viele der Systemfunktionen werden in PROBAnD in den Control-Object-Types gekapselt. In unserem *RoomAutomation*-Beispiel hatten wir z.B. jeweils eigene Objekttypen für die Helligkeits-, Blendungs- und Temperatursteuerung vorgesehen. Will man nur ausgewählte Funktionen in einer Prototypspezifikation repräsentiert sehen, genügt es daher die Control-Object-Types, die nicht betrachtet werden sollen, aus der Anforderungsspezifikation zu eliminieren.

Eine Möglichkeit der Elimination ist, das Verfahren aus dem letzten Abschnitt zu nutzen und alle Control-Object-Types, die nicht betrachtet werden sollen, durch

entsprechende Stubs zu ersetzen. Dies ist dann sinnvoll, wenn zwischen den verbleibenden Objekttypen und den entfernten Teilen eine Wechselwirkung stattfindet, die beim Prototyping berücksichtigt werden soll.

Will man nur getrennte oder unabhängige Funktionalitäten im Prototyp betrachten, so bietet sich als zweite Möglichkeit der Einsatz sog. *Treiber* [May90, S. 449f.] an. Ein Treiber aggregiert dabei nur die relevanten Objekttypen und „simuliert" das notwendige Gesamtsystemverhalten. Abb. 9-8 zeigt eine schematische Darstellung einer Eigenschaftsselektion mit Hilfe eines Treibers.

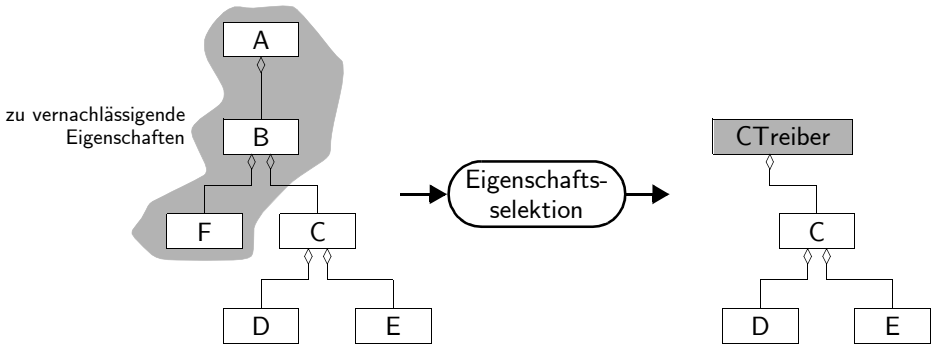


Abbildung 9-8. Strukturelle Abstraktion durch Einsatz eines Treibers

Eine Treibergenerierung erfolgt jeweils für einen einzelnen selektierten Teilbaum, weshalb auch Systemteile, die tiefer im Baum angesiedelt sind (wie der Control-Object-Type C in Abb. 9-8), isoliert einem Prototyping zugeführt werden können.

Ebenso wie für die Stubs aus dem vorhergehenden Abschnitt ist auch für die Treiber eine unterschiedliche Art der Realisierung denkbar. Auch hier kann man strukturelle, passive oder randomisierte Treiber einsetzen. Die Funktionsweise ist analog zu derjenigen der Treiber und auch die Implementierung ist weitgehend analog, daher soll der Code zur Implementierung von Treibern hier nicht explizit angeben werden.

Wichtigster Unterschied besteht in der Erzeugung eines strukturellen Treibers, da hierbei die Erzeugung eines neuen Control-Object-Types an Stelle einer reinen „Umbenennung" des ausgewählten Knotens wie im Falle eines Stubs durchgeführt wird. Der von einem Treiber aggregierte Control-Object-Type kann seinen ursprünglichen Elternknoten, der durch diesen Treiber ersetzt wird, referenzieren („via oo" in SDL bzw. „`-> super`" in HTML, siehe Abschnitt 6.2.3 auf Seite 151) oder auch Signale von diesem konsumieren. Daher muss garantiert werden, dass eine solche Kommunikation weiterhin möglich ist. Insbesondere bedeutet dies, dass in SDL die entsprechenden Kanäle existieren müssen. Es wird daher ein neuer Task und eine neue Strategie für den Treiber synthetisiert. Für diese Strategie wird für jeden vom aggregierten Knoten erzeugten und vom ursprünglichen aggregierenden Objekttyp konsumierten Signaltyp ein entsprechender consumes-Link angelegt. Dasselbe wird in der umgekehrten Richtung für die produces-Links durchgeführt.

Die Umsetzung der passiven und randomisierten Treiber ist dann sehr einfach möglich, indem man sich auf die im strukturellen Treiber synthetisierten produces- und consumes-Links bezieht.

9.2.3 Kopplung von Prototyp und Umgebung

Bereits in Abschnitt 9.1.3 auf Seite 250 sind wir auf die möglichen Arten der Kopplung von reaktivem System (oder dessen Prototyp) und Umwelt (oder deren Simulation) eingegangen. Hier wollen wir die technische Realisierung für eine Kopplung getrennter Systeme, aber auch die Lösung für die Verschmelzung zweier Systeme zu einem geschlossenen System erläutern.

Kopplung getrennter Systeme

Zur Kopplung getrennter Systeme müssen beide zunächst über kompatible Schnittstellen verfügen. Für Experimente im Rahmen der PROBAnD-Methode einigte man sich dazu auf die Verwendung von TCP/IP-Sockets und einem sehr einfachen textuellen Protokoll zum Austausch von Sensordaten und Aktuatorbefehlen über diese Sockets. Die kommunizierenden Komponenten werden dabei über ihren jeweils eindeutigen Instanzennamen identifiziert (siehe auch Abschnitt 6.1.3 auf Seite 138).

Die Details dieser Schnittstelle und deren Realisierung auf Seiten der Umgebung (innerhalb eines physikalischen Testfelds für Experimente im Bereich der Gebäudeautomation) findet man in [Met01].

Die Realisierung auf Seiten des Prototyps erfolgt durch eine benutzerdefinierte C-Bibliothek und die Verwendung von SDL-Remote-Procedure-Calls für den transparenten Zugriff auf die Bibliotheksfunktionen in einem SDL-Modell. Genauere Informationen zu dieser geradlinigen technischen Umsetzung werden in der Arbeit von Queins [Que02, S. 193] und in dem Bericht von Brandt und Schäfer [BrS03] dargelegt.

Will man einen Prototyp an eine Umgebung ankoppeln, die nicht über eine kompatible Schnittstelle zum Prototyp verfügt, so kann man dies bewerkstelligten, indem man eine Komponente zwischenschaltet, die eine Umsetzung der Protokolle vornimmt. In [Met99] wurde z. B. mit Hilfe einer solchen Kopplungskomponente eine Verbindung von PROBAnD-Prototypen mit dem Gebäudesimulator SEMPER [MMZ02] realisiert, der eine Schnittstelle auf der Basis von CORBA besitzt.

Auch bei technisch gleichen Schnittstellen kann das Zwischenschalten einer solchen Kopplungskomponente notwendig werden, wenn die Namen der kommunizierenden Instanzen in reaktivem System und der Umgebung voneinander abweichen (die Identifikation der Kommunikationspartner erfolgt über deren Namen, siehe [MMZ02]). Dies liegt in der Regel an unterschiedlichen Objektstrukturen, die zu ei-

ner anderen Namensvergabe führen (vgl. Abschnitt 6.1.3 auf Seite 138). Um eine solche Namensanpassung vorzunehmen wurde eine Variante der obigen Kopplungskomponente entwickelt. Der Einsatz dieser Komponente wird in [BrS03] erläutert.

Verschmelzung zu einem geschlossenen System

Eine für diese Arbeit interessantere Art der Kopplung ist die Verschmelzung der zu koppelnden Systeme zu einem geschlossenen System, was – wie bereits erwähnt wurde – nur bei identischen Formalismen möglich ist. Dadurch ist eine interne Kommunikation der Sensoren und Aktuatoren (z.B. mittels derselben Remote-Procedure-Calls wie in der obigen Realisierung) möglich und, falls kein Bezug zur Wanduhrzeit besteht, auch eine maximale Beschleunigung des Gesamtsystems zu erreichen.

Ein systematischer und allgemein anwendbarer Ansatz für eine solche Verschmelzung wird von Alanen und Porres in [AlP03] vorgestellt. Für die PROBAnD-Methode ist eine Verschmelzung allerdings einfacher zu realisieren, da die beiden getrennten Systeme jeweils eine baumartige Struktur aufweisen. Unter Zuhilfenahme eines neuen Wurzelknotens lassen sich daher diese beiden Bäume zu einem einzelnen Baum vereinigen. Abb. 9-9 zeigt – wie auch schon bei der Prototypkonstruktion – eine schematische Darstellung dieser Aktivität.

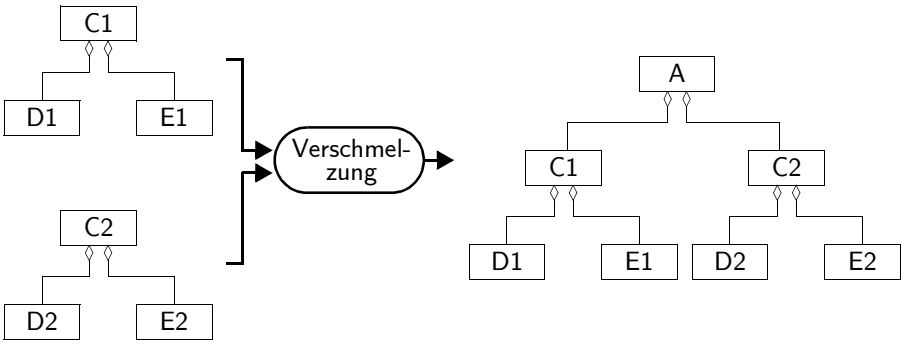


Abbildung 9-9. Verschmelzung zweier Systeme

Da wir die Verschmelzung auch für eine erweiterte Form der Eigenschaftsselektion einsetzen wollen (s.u.), realisieren wir die Verschmelzung in einem Werkzeug in einer allgemeineren Form als oben vorgestellt. An Stelle der Erzeugung eines neuen aggregierenden Knotens (wie am Beispiel von Objekttyp A in Abb. 9-9), gehen wir davon aus, dass eines der zu verschmelzenden Systeme bereits über einen Objekttyp verfügt, welcher der oberste Objekttyp bleiben soll. Dadurch wird es möglich, mehr als zwei Systeme zu verschmelzen ohne eine Kaskadeneffekt auszulösen (bei der ersten Variante der Verschmelzung würde für jeden neu hinzugefügten Teilbaum ein neuer aggregierender Knoten erzeugt). In Abb. 9-10 ist die zweifache Anwendung der so modifizierten Verschmelzung gezeigt.

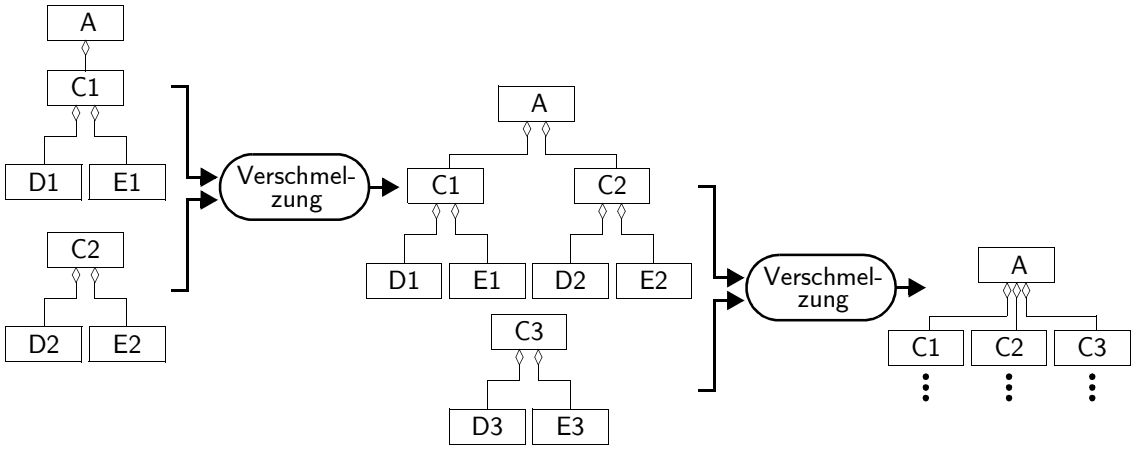


Abbildung 9-10. Gleichwertige Verschmelzung mehrerer Systeme

Zusammen mit der Möglichkeit der Treibergenerierung lässt sich damit nun die Verschmelzung eines reaktiven Systems und dessen Umgebung realisieren. Zunächst erzeugt man für das System $S1$ mit dem obersten Objekttyp C1 einen strukturellen Treiber (also C1Driver). Da der ursprüngliche Objekttyp nirgendwo aggregiert wurde ist dieser strukturelle Treiber quasi „leer“. Dann verschmelzt man dieses so erweiterte System mit dem zweiten System ($S2$, mit dem obersten Objekttyp C2) und erhält die gewünschte Gesamtstruktur, nur dass Objekttyp A (aus Abb. 9-9) nun C1Driver ist.

Auch für den erwähnten Einsatz zur Eigensschaftsselektion lässt sich diese Form der Verschmelzung einsetzen. Angenommen, wir möchten die Funktionalitäten der Objekttypen OT1 bis OT3 eines Gesamtsystems getrennt betrachten. Dann erzeugt man in einem ersten Schritt für alle diese Objekttypen einen Treiber, also OT1Driver bis OT3Driver. Diese getrennten Funktionalitäten lassen sich dann zusammenfügen, wenn man auf einen dieser Treiber (wir nehmen hier OT1Driver) nochmals eine strukturelle Treibergenerierung aufsetzt und damit einen OT1DriverDriver erhält. Dieser entspricht dem Objekttyp A aus Abb. 9-10. Die restlichen OT2Driver und OT3Driver entsprechen den Objekttypen C2 und C3 aus der Abbildung. Das Ergebnis ist dann ein Gesamtsystem, in welchem die Funktionalitäten von OT1 bis OT3 unabhängig voneinander realisiert werden und so in einem Prototyping-Schritt betrachtet werden können.

Die Umsetzung dieser Verschmelzung in einem *Merger*-Werkzeug wird im Folgenden wieder mit Hilfe von AL++ dargestellt. Dazu beginnen wir mit der Definition der Operation `merge()` im abstrakten Artefakttyp RequirementsSpecification:

```
1   /* RequirementsSpecification: */
2   public void merge(RequirementsSpecification rscS2) {
3       Set containedCots =
4        itsObjectStructure.merge(rscS2.itsObjectStructure);
5       itsSystem.merge(rscS2.itsSystem, containedCots);
```

```
6   }
```

Diese `merge()`-Operation wird für die RequirementsSpecification-Instanz des Systems $S1$ aufgerufen und erhält als Aufrufparameter die RequirementsSpecification-Instanz des Systems $S2$. Zunächst erfolgt eine Verschmelzung der Objektstruktur, gefolgt von der Verschmelzung des komplexen Artefakttyps System.

```
1   /* ObjectStructure: */
2   public Set merge(ObjectStructure osS2) {
3       Set containedCots = new HashSet();
4       foreach(ControlObjectType cot in osS2.itsControlObjectType) {
5           if(Utils.setContainsName(itsControlObjectType, cot.name)) {
6               System.err.println("WARNUNG: Control-Object-Type '"+cot+
7                "' von S2 existiert bereits in S1; "+
8                "Gesamtsystem kann Fehlverhalten aufweisen.");
9               containedCots.add(cot);
10          }
11      }
```

Bei der Verschmelzung der Objektstruktur wird zuerst geprüft, ob es bei den zu verschmelzenden Systemen Überschneidungen bzgl. der Objekttypen gibt, d.h. ob in beiden System ein Objekttyp mit identischem Namen existiert (die Hilfsmethode `Utils.setContainsName()` liefert `true`, falls dies der Fall ist). Es wird eine Warnung ausgegeben, da man die korrekte Funktionalität des Gesamtsystems bei der Verschmelzung von MSR-System und Umgebungssimulator dann nicht garantieren kann. Denn Objekttypen mit gleichem Namen implizieren in einem solchen Falle jeweils unterschiedliche Realisierungen (so unterscheidet sich die Realisierung des TempSens-Control-Object-Types von *RoomAutomation* deutlich von der Realisierung des Objekttyps im zugehörigen Gebäudesimulator).

Dem Entwickler obliegt es daher solche kritischen Stellen zu prüfen und eventuell die Objekttypen in einem der Systeme sinnvoll umzubenennen. Man könnte z.B. an alle Namen der Simulator-Objekttypen ein „`_SIM`“ anhängen.

Bei dem Einsatz der Verschmelzung zur Eigenschaftsselektion kann es allerdings durchaus vorkommen, dass Objekttypen mehrfach vorkommen (wenn z.B. ein generischer Sensor wie z.B. Contact in mehreren Teilbäumen Verwendung findet). Diejenigen Objekttypen, die doppelt vorkommen, werden daher in der Menge `containedCots` abgelegt, welche später genutzt wird, um solche Objekttypen nicht doppelt in das komplexe Artefakt System aufzunehmen.

```
12      ControlObjectType rootS1 = topLevelControlObjectType;
13      ControlObjectType rootS2 = osS2.topLevelControlObjectType;
14      String instName = rootS2.name.toLowerCase();
```

```
15      if(instName.length() >= 3)
16          instName = instName.substring(0, 3)+"1";
17      else
18          instName = instName+"1";
19      Instantiation inst = new Instantiation(instName);
20      inst.instantiatedControlObjectType = rootS2;
21      rootS1.aggregatedInstantiation += inst;
22      return containedCots;
23  }
```

Von Zeile 12 bis Zeile 21 findet sich die eigentlich interessante Stelle bei der Verschmelzung, da hier die beiden Teilbäume zu einem gemeinsamen Baum zusammengefasst werden. Dazu stellt man zunächst jeweils die Wurzel der Teilbäume fest (`rootS1` und `rootS2`). Nun muss eine Instanziierung von `rootS2` erzeugt werden und als aggregierte Instanz in `rootS1` eingehängt werden. Dazu erzeugt man eine Instanz des Typs Instantiation. Als Name verwenden wir hier einfach die ersten drei Buchstaben des Objekttypnamens in Kleinbuchstaben (eine Prüfung auf Namenskollisionen lassen wir in dieser Darstellung wie auch bereits bei der Einführung der Stubs weg).

Nach einer Zusammenführung der Systemstruktur muss nun noch der komplexe Artefakttyp System angepasst werden.

```
1   /* System: */
2   public void merge(System sysS2, Set containedCots) {
3       foreach(ControlObjectTypeConfiguration cotc in
4        sysS2.itsControlObjectTypeConfiguration) {
5           if( !containedCots.contains(cotc.itsControlObjectType) )
6               itsControlObjectTypeConfiguration += cotc;
7       }
8   }
```

Hierzu werden alle ControlObjectTypeConfigurations, die im System $S2$ neu definiert werden hinzugefügt. Da wir in der Menge `containedCots` bereits alle „doppelten“ Objekttypen gespeichert haben, ist dies sehr einfach zu realisieren.

Neben der oben dargestellten Änderung der Struktur ist für die Verschmelzung zweier Spezifikationen zu einer vollständigen Gesamtspezifikation auch die Zusamenführung der Anforderungen notwendig. Dies wird durch die Hinzunahme zweier neuer `merge()`-Operationen und deren Aufruf in der `merge()`-Methode der RequirementsSpecification wie folgt ermöglicht:

```
itsNeeds.merge(rscS2.itsNeeds, itsTaskList);
itsTaskList.merge(rscS2.itsTaskList);
```

Die Operation zum Verschmelzen der Needs erfolgt im komplexen Artefakttyp Needs:

```
1   /* Needs: */
2   public void merge(Needs needsS2, TaskList taskListS1) {
3       foreach(Need need in needsS2.itsNeed) {
4           if( !Utils.setContainsName(itsNeed, need.name) ) {
5               itsNeed += need;
6               taskListS1.itsNeed += need;
7           } else {
8               System.err.println("WARNUNG: Need '"+need+
9               "' von S2 existiert bereits in S1; wird uebersprungen.");
10          }
11      }
12  }
```

Für alle Needs des Systems $S2$ wird geprüft, ob ein Need desselben Namens auch im System $S1$ existiert. Ist dies nicht der Fall, so wird der Need in die Liste der Needs von $S1$ aufgenommen. Entsprechend erfolgt die Eintragung dieses zusätzlichen Needs in die Taskliste. Existiert ein Need mit solchem Namen bereits, dann wird wie im obigen Fall der Objekttypen eine Warnmeldung ausgegeben.

Analog erfolgt das Verschmelzen der Tasks im komplexen Artefakttyp TaskList:

```
1   /* TaskList: */
2   public void merge(TaskList taskListS2) {
3       foreach(Task task in taskListS2.itsNeed) {
4           if( !Utils.setContainsName(itsTask, task.name) ) {
5               itsTask += task;
6           } else {
7               System.err.println("WARNUNG: Task '"+task+
8                "' von S2 existiert bereits in S1; wird uebersprungen.");
9           }
10      }
11  }
```

Bei der Interpretation der Warnmeldungen ist wieder zu unterscheiden, ob es sich um eine Verschmelzung getrennter Systeme (zur Realisierung der Kopplung von MSR-System und Umwelt, erster Fall von oben) oder um die Verschmelzung von Teilen desselben Systems (zur Eigenschaftsselektion) handelt.

Im ersten Fall bedeutet die Existenz von Anforderungen mit identischen Namen, dass es nach der Verschmelzung zu einer fehlerhaften Spezifikation kommt. Die Begründung ist dieselbe wie bei den Objekttypen, weshalb auch wie dort erklärt wurde zunächst eine Umbenennung der Anforderungen erfolgen muss.

Im zweiten Fall ist es nicht zulässig Needs und Tasks, die in beiden Teilsystemen identisch sind, doppelt in die Liste der Anforderungen aufzunehmen, da diese jeweils die Anforderungen desselben Gesamtsystems widerspiegeln und so eine unerwünschte Redundanz in der Spezifikation entstehen würde.

Ein Problem taucht bei obiger Vorgehensweise allerdings auf. Existieren neben den oben erwähnten Artefakten auch andere Artefakte mit identischem Namen (z.B. Signaltypen), so müssen diese als identische Artefakte interpretiert werden (ein Signaltyp muss z.B. von einem Teilsystem zum anderen versendet werden können). Auf Ebene der abstrakten Artefakte handelt es sich trotz dieser Namensgleichheit allerdings um zwei unterschiedliche Instanzen desselben abstrakten Artefakttyps, da zwei getrennte Spezifikationen als Eingabe dienten. Daher sind die Verfolgbarkeitsrelationen zwischen diesen Teilmodellen durch die obigen Verschmelzungsalgorithmen noch nicht vollständig rekonstruiert. Abb. 9-11 zeigt einen Fall für den Signaltyp stA.

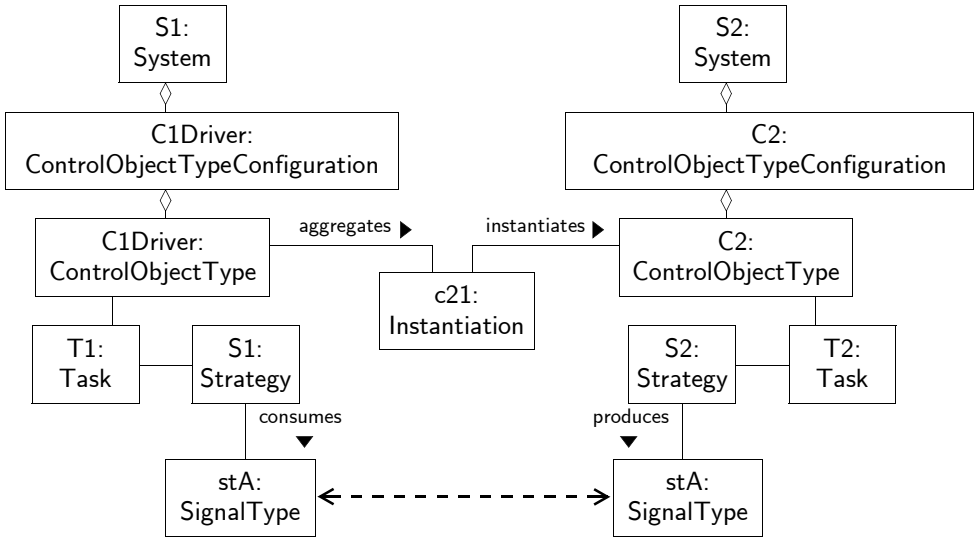


Abbildung 9-11. Problem bei der Verschmelzung zweier Systeme (abstrakte Artefakte)

Um diese Situation auf Ebene der abstrakten Artefakte zu korrigieren, müssen alle solchen Artefakt-Instanzen identifiziert werden und jeweils zu einer einzigen Instanz zusammengefasst werden (in Abb. 9-11 ist dies durch die gestrichelte Linie angedeutet). Eine solche Zusammenfassung ist allerdings nicht einfach, da neben den Artefakt-Instanzen auch alle Links, die solche Instanzen verbinden (z.B. produces), und eventuell alle über diese Links verbundenen anderen Artefakt-Instanzen (z.B. FormalParameter) einer solchen Prozedur unterzogen werden müssten. Leider lässt sich dies nicht generisch realisieren, da einige der Artefakte (wie z.B. die Attribute oder die Instanzen vom Artefakttyp Signal) durchaus mehrfach mit demselben Namen im Modell auftauchen dürfen (z.B. falls verschiedene SendActions existieren, die ein Signal mit jeweils unterschiedlichen aktuellen Parametern versenden).

Der dokumentzentrierte Ansatz bietet für dieses Problem eine sehr elegante Lösung: Nachdem das „Mergen“ durchgeführt wurde, erzeugt man zunächst durch Einsatz des Unparsers PROBAnD-Dokumente in der konkreten Syntax. Nun werden diese Dokumente wieder geparsed und man erhält als Resultat eine Produktmodellinstanz (abstrakte Artefakte), welche keine mehrfachen Instanzen für die relevanten identischen Artefakte beinhaltet. Der Grund ist, dass beim Parsen über eine Namensgleichheit die Identität von Artefakten beurteilt wird (siehe Abschnitt 5.1.2 auf Seite 91). Das Auftauchen des Signaltyps stA an zwei Stellen wird damit korrekt als die Existenz einer einzelnen Instanz des abstrakten Artefakttyps SignalType interpretiert.

Zusammenfassend hier nochmals die Schritte und die jeweils eingesetzten Werkzeuge, um zwei Systeme $S1$ und $S2$ mit jeweils C1 und C2 als Wurzelknoten zu koppeln:

1. Erzeugung eines Treibers C1Driver für C1 von $S1$. Ergebnis ist ein modifiziertes System $S1'$ (Clipper)
2. Verschmelzen des modifizierten Systems $S1'$ und des Systems $S2$ zu einem Gesamtsystem S (Merger)
3. Generieren der PROBAnD-Dokumente (konkrete Artefakte) aus den abstraken Artefakten von S (Unparser)
4. Parsen der Dokumente aus dem vorhergehenden Schritt. Ergebnis sind die gewünschten (und konsistenten) abstrakten Artefakte des Zielsystem S' (Parser)

Man hat hier also eine komplexe Aktivität durch die Komposition (oder Hintereinanderausführung) mehrerer einfacher Aktivitäten realisiert.

9.3 Automatisiertes Prototyping

Die letzten Abschnitte zeigten, wie man die Eigenschaftsselektion und die anschließende Konstruktion von Prototypen automatisieren kann. Neben einer solchen Automatisierung bei der Erstellung von Prototypen ist es nun auch möglich, die Anwendung der Prototypen, also das Prototyping im weiteren Sinne, zu automatisieren.

Dazu werden wir in diesem Abschnitt drei Ansätze vorführen. Wir beginnen mit der Unterstützung einfacher Tests, zeigen dann die Möglichkeiten einer dynamischen Analyse auf und werden schließlich die Vision einer vollständig automatisierten Prototyping-Umgebung skizzieren.

9.3.1 Testen

Ein wichtiges Qualitätskriterium eines Tests ist dessen *Überdeckung*, also der Anteil der getesteten Aspekte (wie z.B. der geprüften Verhaltens-Traces) an allen Aspekten (wie z.B. an allen möglichen Traces des Systems). Bezüglich dieses Kriteriums bieten die deterministischen endlichen Automaten aus Definition 6-1 auf Seite 151 die schöne Eigenschaft, dass eine vollständige Überdeckung erreicht wird, wenn man jede Transition einmal getestet hat (siehe dazu auch [BrH93, S.339]).

Für die in der PROBAnD-Methode spezifizierten Systeme, welche aus parallelen Prozessen bestehen, welche jeweils durch einen erweiterten Automaten beschrieben werden, kann eine solche vollständige Überdeckung wie wir in Abschnitt 6.2.3 auf Seite 151 bereits andeuteten nicht erreicht werden. Dazu müsste man alle möglichen Kombinationen von Zuständen und Transitionen in allen diesen Prozessen überprüfen [BrH93, S.341]. Dass dies alleine durch die Definition einer Integer-Variable in einem der EFSMs ein unmögliches Unterfangen wird, ist offensichtlich. Eine mögliche Lösung ist die Abstraktion von solchen unendlichen Wertebereichen durch die Technik der „Prädikatsabstraktion“ (siehe z.B. [Var04, S.95f.]). Eine andere Möglichkeit ist das Reduzieren der Tests auf *relevante* Szenarien.

Szenariobasiertes Testen

Beim *szenariobasierten Testen* beginnt man zunächst mit den normalen Szenarien und überprüft dann im weiteren Verlauf auch Ausnahme- oder Extremsituationen (siehe [BrH93, S.341f.]) und eventuelle unerwünschte Verhältnisse. Ein Testfall entspricht einem einzelnen erwünschten oder unerwünschten Ablauf des Systems, wobei in dem Testfall nur die Kommunikation zwischen dem zu testenden System (*Testobjekt*, „system under test“, SUT) und dessen Umgebung (*Testumgebung*) spezifiziert wird. Die Reihenfolge der Nachrichten in einem Testfall kann jedoch auch von der Kommunikation innerhalb der Testumgebung oder innerhalb des Testobjekts abhängen (vgl. z.B. [BBJ03]).

Auf dem Gebiet des szenariobasierten Testens gibt es unzählige Forschungsbeiträge, die zum Teil einen hohen Grad an Automatisierung aufweisen. Im Kontext von SDL sei hier auf die in Telelogic Tau integrierte Testumgebung verwiesen (vgl. [Dol03, S.111ff.]). Auch die Arbeiten von Baker et al. sind erwähnenswert, da das dort vorgestellte *ptk*-Werkzeug [BBJ03] über die Funktionalität von Telelogic Tau hinausgeht. Dieses Tool generiert ausgehend von MSCs die entsprechenden *Testskripten* in der Testnotation *TTCN*-2 (*Tree and Tabular Combined Notation* [Wil98]), wobei versucht wird, ein Optimum aus der Zahl der zu erzeugenden Test-Skripten und der Testüberdeckung zu erreichen. Die TTCN-Skripten können als Eingabe in ein entsprechendes Testwerkzeug (z.B. Telelogic Tau TTCN Suite) dienen und dann vollautomatisch abgearbeitet werden.

Der allgemeine Aufbau einer Testumgebung inklusive des Testobjekts, welches in unserem Falle den PROBAnD-Prototypen entspricht, ist in Abb. 9-12 gezeigt.

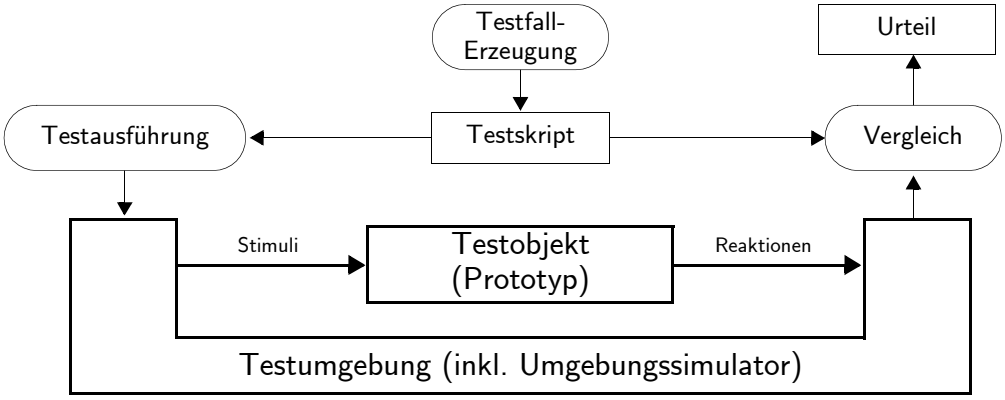


Abbildung 9-12. Aufbau einer Testumgebung mit Testobjekt

Ausgehend von einem oder mehrerer Testskripten wird während der Testausführung die Testumgebung so stimuliert, dass die spezifizierten Nachrichten zu dem Testobjekt gesendet werden und die vom Testobjekt gelieferten Nachrichten aufgezeichnet und mit dem im Skript geforderten Antworten verglichen werden. In Abhängigkeit von dem Ausgang des Vergleichs erfolgt die Beurteilung des Tests in „pass“ (bestanden) oder „fail“ (durchgefallen).

Da die Umgebung eines reaktiven System normalerweise ein komplexes Verhalten aufweist, das sich nur schwer durch statische Skripten beschreiben lässt (vgl. Abschnitt 9.1.3 auf Seite 250), kann durch das Testen einzelner Testläufe nur ein isoliertes Systemverhalten überprüft werden. Wie wir schon erläutert hatten, sind für realistische Tests daher Umgebungssimulatoren (als Teil der Testumgebung) unabdingbar. Sinnvoll kann daher nur eine Kombination der skriptbasierten Testtechniken mit solch einem Umgebungssimulator sein. Zum Test eines Gebäudeautomationssystems könnte solch ein Skript z.B. einfaches Personenverhalten und den Wetterverlauf beinhalten, wohingegen die Temperatur in den Räumen und andere physikalische Größen vom Gebäudesimulator berechnet würden.

Natürlich sollte auch der Umgebungssimulator getestet werden. In einem solchen Fall nimmt dieser die Rolle des Testobjekts ein. Für Verfahren zur Beurteilung der Güte von Gebäudesimulatoren sei auf die Arbeiten von Zimmermann [Zim02] und Riegel [Rie02][Rie04] verwiesen.

Ein weiteres Problem besteht in der Beurteilung, ob ein gewünschtes Systemverhalten eingetreten ist, da man bei den von uns betrachteten Systemen nicht nur atomare Ereignisse als Reaktion auf Stimuli betrachtet, sondern auch zeitliche Verläufe von Messgrößen (wie z.B. der Temperatur). Wir hatten bei der Illustration der PROBAnD-Methode bereits eine Temperaturkurve (Abb. 6-8 auf Seite 146) gezeigt. Der zeitliche Verlauf einer solchen Kurve entscheidet oftmals über die Korrektheit

eines Regelalgorithmus, wenn z.B. keine Überschwinger auftauchen dürfen. Wir werden zum Ende dieses Kapitels in Abschnitt 9.3.3 eine Möglichkeit aufzeigen, wie ein solcher Vergleich für einen speziellen Typ von Testläufen realisiert werden kann.

Randomisierte Tests

Eine alternative Erzeugung von Testfällen stellt der *Zufallstest* dar, bei welchem aus den Wertebereichen der Eingabedaten zufällige Werte ausgewählt werden [Lig02, S.203f.]. Anders als z.B. bei Äquivalenzklassentests wird der Eingabewertebereich nicht partitioniert, sondern die Werte werden entsprechend einer vorgegebenen Wahrscheinlichkeitsverteilung ausgewählt.

Trotz der nicht-deterministischen Auswahl der Testfälle (wenn wir mal annehmen, dass der Pseudo-Zufallsgenerator mit einem quasi zufälligen Wert, z.B. der Systemzeit, initialisiert wird), haben Untersuchungen gezeigt, dass Zufallstests nicht deutlich schlechter als deterministische Verfahren sind [Lig02, S.203]. Der Vorteil von Zufallstests liegt in dem geringeren Aufwand, der zur Erstellung der Testfälle nötig ist. Allerdings lassen sich die gewünschten Reaktionen des Systems nicht ohne weiteres automatisch (oder gar per Zufall) bestimmen, weshalb die Auswertung der Testläufe eventuell manuell erfolgen muss oder ein entsprechender Aufwand für die Erstellung von *Testorakeln* (siehe z.B. [RAM92]) aufgewendet werden muss. Wie bei allen Qualitätssicherungsmaßnahmen sollte daher auch hier gelten, dass Zufallstests am Besten ergänzend zu anderen Testtechniken eingesetzt werden.

Für die von uns betrachteten reaktiven Systeme ist der Zufallstest – wie auch bereits der Szenariotest – nicht in einer solchen einfachen Form anwendbar. Wenn auch manche Stimuli durch Zufallswerte ersetzt werden können, gilt dies für den Großteil der Eingaben in das reaktive System leider nicht. Auch dies liegt wieder an der bereits erwähnten Komplexität der Umgebung. So stellt z.B. eine über die Zeit zufällig schwankende Außentemperatur im Intervall [-10°C, 30°C] kein sinnvolles Testszenario dar, da selbst in Extremsituationen ein solcher Fall in der Realität nie auftreten würde.

Für einige ausgewählte Aspekte (wie z.B. das bereits erwähnte Personenverhalten) kann eine solche zufällige Testfallgenerierung aber dennoch sinnvoll sein. Für die Erzeugung zufälliger Eingaben können wir auf die bereits in Abschnitt 9.2.1 auf Seite 258 vorgestellte Technik zur automatischen Erstellung randomisierter Stubs zurückgreifen. Wir müssen nur für jeden Sensor des Testobjekts, der mit zufälligen Werten stimuliert werden soll, einen entsprechenden Stub erzeugen (unter Angabe der gewünschten zeitlichen Verteilung und des Wertebereichs).

Zur Illustration dieser Technik wählen wir unser *RoomAutomation*-Beispiel aus Abschnitt 6.3 auf Seite 167. In Ergänzung zu der dort vorgestellten Spezifikation nutzen wir den Bewegungsmelder MotionSens, um die Raumtemperatur nur im Falle einer Belegung des Raumes auf Solltemperatur zu halten. Dies soll eine Energie-

einsparung bringen, da dann in einem nicht belegten Raum die Temperatur abgesenkt werden kann (man kann sich dies als eine Realisierung eines verallgemeinerten Needs N2 aus Tabelle 6-1 auf Seite 139 vorstellen). Der MotionSens liefert dabei eine „1“ so lange sich jemand im Raum bewegt und eine „0“, falls keine Bewegung mehr festgestellt werden kann. Da die Bewegung kein direktes Maß für die Belegung des Raumes ist (es könnte sein, dass eine Person ruhig am Schreibtisch sitzt), wird die Raumtemperatur erst nach einer gewissen Verzögerung Δt_{occ} abgesenkt.

In dieser so ergänzten Spezifikation ersetzten wir den MotionSens durch einen randomisierten Stub, womit wir ein zufälliges Personenverhalten nachbilden konnten. Die Ergebnisse eines Simulationslaufs von insgesamt 21,6 Stunden Modellzeit sind in Abb. 9-13 gezeigt (die Außentemperatur betrug konstant 5 Grad).

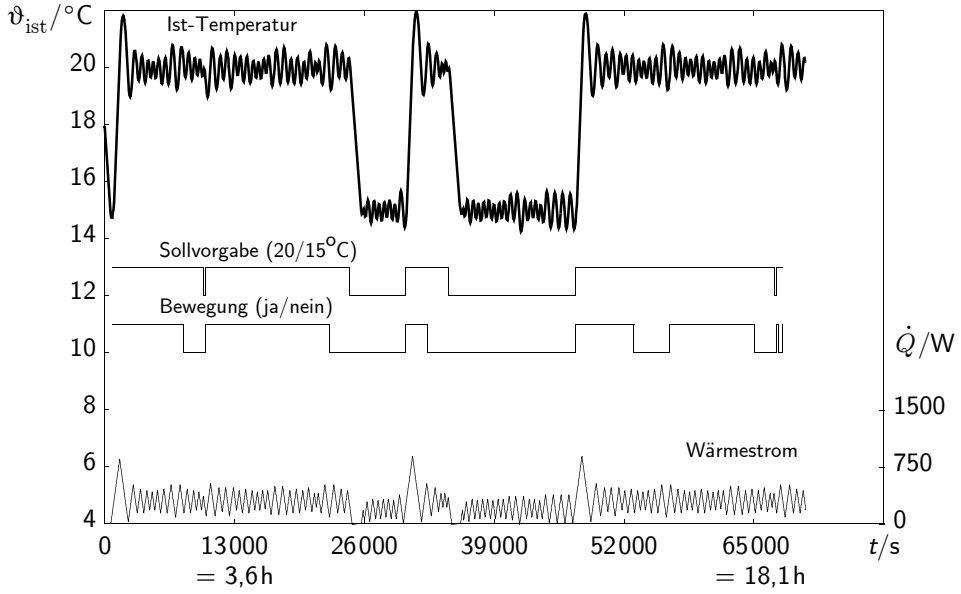


Abbildung 9-13. Ergebnis eines randomisierter Laufs

Wie man erkennt, wird der Raum beheizt, sobald eine Person eintritt (also ein Bewegungsereignis auftritt) und die Temperatur abgesenkt, wenn innerhalb von $\Delta t_{\text{occ}} = 1950\,\text{s}$ (32,5min) nach der letzten Bewegung keine neue Bewegung festgestellt werden konnte. Als Intervall, aus dem gleichverteilt der nächste Zeitpunkt eines Bewegungsereignisses (newMotion-Signal) „gezogen“ wird, wählten wir $[0\,\text{s}, 5850\,\text{s}] = [0\,\text{min}, 97{,}5\,\text{min}]$, wobei mit gleicher Wahrscheinlichkeit zu diesem Zeitpunkt entweder die „1“ (für Bewegung) oder die „0“ (für keine Bewegung) gezogen wird.

Um den Energieverbrauch beurteilen zu können ist der Wärmestrom $\dot{Q}$ der Heizkörper (in der Einheit Watt, siehe [LJK94, S.590]) dargestellt. Zur Protokollierung dieser Größe, verwendeten wir anders als bei der Vorstellung des *RoomAutomation*-Beispiels in Abschnitt 6.3 auf Seite 167 nicht den Gebäudesimulator von Riegel,

sondern einen mittels der PROBAnD-Methode spezifizierten Simulator von Rehm (siehe [Reh04]). Damit wurde eine einfache automatische Instrumentierung möglich.

Zusätzlich konnte dadurch das reaktive System und der Simulator verschmolzen werden und bei der Berechnung der Messwerte die maximale Geschwindigkeit erreicht werden (die Berechnung benötigte nur 4s auf einem AMD Athlon PC mit 1,67GHz unter Windows XP). Der Kurvenverlauf weicht von dem in Abb. 6-8 auf Seite 146 ab, da im Simulator von Rehm außer der Trägheit der Heizkörperventile andere physikalische Parameter für den simulierten Raum eingestellt waren (z.B. dünnere Wände). Für die folgenden Betrachtungen sind diese Unterschiede allerdings nicht weiter relevant.

Zur Überprüfung der Energieeinsparung müssen wir die eingesetzte Energie bei der „intelligenten" Regelung mit Abschaltung dem Energieverbrauch gegenüberstellen, der ohne eine solche Abschaltung nötig gewesen wäre. Der Energieverbrauch ist proportional zu der insgesamt abgegebenen Wärmemenge Q der Heizkörper (in der Einheit Joule, siehe [LJK94, S.585]) und lässt sich durch Integration der Wärmeströme erhalten zu:

$$Q = \int_t \dot{Q}\,dt \qquad \textbf{(Gleichung 9-1)}$$

Da wir uns bei der Realisierung der Gebäudesimulatoren nicht im kontinuierlichen sondern im diskreten Bereich bewegen, wird aus dem Integral eine Summe. Für das obige Experiment erhielten wir ein $Q = 2482\text{kJ}$ für die „intelligente" Regelung und ein $Q = 2700\text{kJ}$ für die einfache Regelung.

Damit könnte man nun die weitere Vermutung anstellen, dass bei einer solchen zufälligen Belegung des Raumes eine passende Einstellung der Verzögerungszeit Δt_{occ} einen relevanten Einfluss auf den Energieverbrauch hat (für die einfache Regelung wäre $\Delta t_{\text{occ}} = \infty$). Zum Ende dieses Kapitels werden wir diese Hypothese weiter untersuchen.

Man sieht an diesem Beispiel allerdings auch die Grenzen des Zufallstests, da wirklichkeitsnähere Szenarien den natürlichen Arbeitsablauf (z.B. längere Aufenthalte im Raum, Mittagspause, etc.) berücksichtigen und damit vermutlich zu anderen Ergebnissen führen würden.

Konstruktive Maßnahmen zur Testunterstützung

Wie bereits erläutert, kann mit Tests nie eine vollständige Überdeckung eines Systems von mindestens moderater Größe erreicht werden. Szenariobasierte Tests sind selektiv und auch die randomisierten Vertreter wählen immer nur Ausschnitte aus allen Aspekten, die es zu prüfen gilt.

Man kann daher Fehler im endgültigen System, die sich erst durch die komplexen Wechselwirkung von Prozessen als Fehlverhalten manifestieren, nie ganz ausschließen. Ein Teil solcher Probleme lässt sich zwar prinzipiell durch eine Ereichbarkeitsanalyse identifizieren (siehe dazu auch Constraint $CO2$ in Abschnitt 8.1.2 auf Seite 192), aber bei sehr großen Systemen skaliert dieser formale Ansatz nicht unbedingt, auch wenn es erfolgreiche Anwendungen gibt (vgl. [Lig02, S. 383]). Desweiteren ist eine formale Verifikation der Spezifikation während der Analyse- (oder Requirements-Engineering)-Phase schwierig, da kein formales Dokument existiert, gegen welches eine solche Verifikation durchgeführt werden könnte (die Problembeschreibung besteht lediglich aus einer Liste nicht-formal beschriebener Needs, siehe Abb. 9-1 auf Seite 242 und Abschnitt 6.1 auf Seite 131).

Es ist somit notwendig, Mechanismen im ausgelieferten System zu installieren, welche mit eventuellen Fehlern umgehen und dadurch die Robustheit (siehe Abschnitt 7.1.1 auf Seite 177) des Software-Produkts sicherstellen (vgl. [BrH93, S. 341]). In unsere Augen kann eine solche Robustheit auch bereits für Prototypen relevant werden. Insbesondere wenn man Prototypen reaktiver Systeme über einen längeren Zeitraum in einer Testumgebung betreibt (wie wir es z. B. für die Evaluierung von Gebäudeautomationssystemen durchführten, siehe [Met01, S. 43ff.]), kann man es sich nicht erlauben, dass der Prototyp während eines solchen Testbetriebs ausfällt und damit wertvolle Zeit verloren geht.

Ein solcher Ausfall kann bei den von uns betrachteten und in der PROBAnD-Methode spezifizierten Systemen insbesondere dadurch zu Stande kommen, dass ein Prozess sich in einem Zustand befindet, in welchem er das gerade ankommende Signal nicht empfangen kann (vgl. Abschnitt 9.1.2 auf Seite 246).

Eine solche Situation kann selbst dann eintreten, wenn Constraint $CS3$ aus Abschnitt 8.1.2 auf Seite 192 eingehalten wird. Denn dieser Constraint verlangt nur, dass ein Signal prinzipiell empfangen werden kann, nicht aber, dass es immer auch konsumiert wird. Man könnte nun prinzipiell dieses Constraint so erweitern, dass in einem gültigen Modell alle möglichen Signalempfänge von jedem Zustand aus spezifiziert werden müssten. Dies ist allerdings nicht unbedingt sinnvoll, weil eine Stärke der zustandsbasierten Ansätze gerade darin liegt, die Komplexität dadurch zu reduzieren, dass nur die relevanten Transitionen beschrieben werden.

Mit Hilfe unseres Automatisierungsansatzes können wir aber eine andere Lösung anbieten, welche ohne eine Einschränkung des Modellierers durchgeführt werden kann: Man erzeugt die nicht spezifizierten Signalempfänge automatisch und protokolliert diese. Die Implementierung ist analog zu der Implementierung der Stubs und Treiber (siehe Abschnitt 9.2 auf Seite 258), weshalb wir hier davon absehen wollen, den AL++-Code erneut darzulegen.

Nach der Ausführung des Systems können dann an Hand der Protokolldatei die Laufzeitfehler analysiert, mögliche Fehlerursachen identifiziert und bei Bedarf entsprechend korrigiert werden.

Es sei angemerkt, dass die von uns verwendete SDL Umgebung Telelogic Tau bereits über mächtige Protokollierungsfunktionen verfügt, welche auch verlorene Signale aufzeichnet. In einem Einsatz auf Rechnern mit kleinem Speicher (wie z.B. Embedded PCs) kann ein solches ausführliches Protokoll allerdings schnell den verfügbaren Platz sprengen. Neben der hohen Speicherkomplexität weist diese Instrumentierung auch eine hohe Zeitkomplexität auf, da jede SDL-Anweisung mitprotokolliert wird (unsere Messungen haben gezeigt, dass diese Protokollierung die Ausführungszeit bis zu einem Faktor von 22 unter HP-UX und 68 unter Windows verlangsamen kann). Es ist daher angebracht nicht erst am Ende aus einem Gesamtprotokoll die relevante Information herauszufiltern, sondern sich bereits beim Aufzeichnen auf die relevanten Daten zu beschränken.

9.3.2 Prototyping für die dynamische Analyse

Neben des Einsatzes von Prototypen für die Validierung und für das gezielte Testen zum Zwecke der Verifikation können mit Prototypen auch *dynamische Analysen* (siehe z.B. [Tra93, S.155f.]) durchgeführt werden. Dabei werden auch interne Eigenschaften des Testobjekts neben dessen Außenverhalten überprüft. Die Analyseergebnisse werden dabei auf Grund einer oder mehrerer Ausführungen des Prototyps und der Messung der relevanten Laufzeitgrößen berechnet.

Zur Messung der Größen ist eine *Instrumentierung* des Testobjekts notwendig, welche die relevanten Daten nach außen (an die Testumgebung) liefert. Ein Beispiel für eine Instrumentierung von Modellen hatten wir bereits in Abschnitt 6.3 auf Seite 167 zur Illustration unseres Automatisierungsansatzes vorgeführt. Auch die konstruktive Maßnahme zur Protokollierung von nicht empfangenen Signalen aus dem vorhergehenden Abschnitt hatte einen ähnlichen Charakter. Im weiteren Verlauf werden wir ein weiteres Beispiel für eine solche Instrumentierung vorstellen.

Zunächst sollte aber darauf hingewiesen werden, dass durch jede Form der Instrumentierung das Prüfobjekt im Allgemeinen verändert wird (vgl. [Tra93, S.156]), was zu einer Veränderung des Verhaltens gegenüber dem Verhalten des Ursprungssystems führen kann. Es handelt sich bei einer Instrumentierung also um eine Modelltransformation vom Typ SM (vgl. Abschnitt 2.2.2 auf Seite 19).

Durch eine Automatisierung der Instrumentierung kann aber zumindest sichergestellt werden, dass keine Fehler in das System eingeführt werden. Desweiteren ließe sich durch einen Vergleich von Systemen mit einem unterschiedlichen Grad an Instrumentierung (von „keine Instrumentierung“ bis „voll instrumentiert“) auch der Einfluss einer Instrumentierung beurteilen. Zum Beispiel könnte man untersuchen,

wie sich eine Instrumentierung auf das zeitliche Verhalten (z.B. die Reaktionszeit) des Systems auswirkt.

Von Queins [Que02, S.201ff.] und Queins et al. [QST99] wird ein Verfahren zur dynamischen Analyse von SDL-Modellen vorgestellt, bei welchem die Analyse der internen Signalkommunikation auf der Basis der in der Telelogic Tau SDL Suite integrierten Protokollierungsfunktion durchgeführt wird. Nachteilig ist hierbei, dass neben der großen Datenmenge, die aufgezeichnet werden muss, diese Datenmenge auch das Zeitverhalten des Prototyps deutlich beeinflusst, da viele – zum Teil irrelevante – Ein-/Ausgabeoperationen durchgeführt werden müssen. Eine gezielte Instrumentierung, wie wir sie weiter oben bereits andeuteten, ist hier sinnvoller. Wir wollen allerdings nicht die Analyse von Queins et al. nach-implementieren, sondern zur Illustration eine dynamische Analysemethode vorstellen, die einen größeren Gebrauch von dem Produktmodell der PROBAnD-Methode macht.

Dynamische Analyse der Anforderungsüberdeckung

Wie wir bereits erwähnten, ist die Testüberdeckung ein wichtiges Kriterium für einen Testfall. Für die typischen Formen der Testüberdeckung, wie z.B. die Anweisungs- oder Zweigüberdeckung, sei auf die Literatur verwiesen (als Beispiel [Lig02, S.79ff.] oder [Tra93, S.215ff.]). Ein weiteres Überdeckungsmaß könnte die Beurteilung der in einem Testfall abgedeckten Benutzer- oder Entwickleranforderungen sein (Egyed et al. stellen in [EgG02] einen verwandten Ansatz für die Anforderungsüberdeckung in Java-Code vor). Insbesondere, wenn der Test im Rahmen einer Validierungsaktivität durchgeführt wird, könnte man so Rückschlüsse auf die vom Nutzer wirklich evaluierten Anforderungen ziehen und bei Bedarf zusätzliche Testfälle mit diesem durchspielen, um insgesamt eine bessere Abdeckung zu erreichen.

Eine Instrumentierung unserer PROBAnD-Modelle zu diesem Zweck gestaltet sich äußerst einfach, da für jede Transition lediglich die realisierte Strategie und von dort der realisierte Task ausfindig gemacht werden muss (siehe Produktmodell aus Abb. 6-10 auf Seite 150 und Abb. 6-12 auf Seite 154). Dieser Task liefert uns dann auch die überdeckten Benutzeranforderungen (Needs).

Der AL++-Code zur Realisierung einer solchen Instrumentierung wird wie folgt in der `instrument()`-Operation des abstrakten Artefakttyps Task definiert. Dabei muss zur vollständigen Instrumentierung die `instrument()`-Operation nur für alle Tasks der TaskList aufgerufen werden.

```
1   /* Task: */
2   public void instrument() {
3       Strategy s = realizingStrategy;
4       Set needs;
5       foreach(Transition tr in s.realizingTransition) {
6           Integer lastActionNbr = 0;
```

```
7           foreach(Action act in tr.effectedAction) {
8               if(act.actionNbr > lastActionNbr) {
9                   lastActionNbr = act.actionNbr;
10              }
11          }
12          needs = this.computeRealizedNeeds(new HashSet());
13          Action rca = new RemoteCallAction("InspectionAction");
14          rca.description = "printIt('needs: "+needSet+
15                            " task: "+this.name+"', now)";
16          rca.actionNbr = lastActionNbr + 1;
17          tr.effectedAction += rca;
18      }
19  }
```

Die Instrumentierung beginnt mit der Feststellung der realisierenden Strategie. Dann wird für jede Transition, welche den jeweiligen Task realisiert, eine neue Aktion an das Ende der Transition eingefügt. Dazu wird zunächst die Nummer der letzten Aktion bestimmt.

Nur die Operation `computeRealizedNeeds()` des Artefakttyps Requirement bleibt noch zu klären:

```
1   /* Requirement: */
2   public Set computeRealizedNeeds(Set needs) {
3       if(this.type == Need) {
4           needs.add(this);
5       } else {
6           Task t = (Task)this;
7           foreach(Requirement req in t.realizedRequirement) {
8               req.computeRealizedNeeds(needs);
9           }
10      }
11      return needs;
12  }
```

Auch diese Realisierung ist sehr einfach. Beginnend bei der aktuellen Requirement-Instanz werden die realizedBy-Relationen nach oben zur Wurzel hin verfolgt und alle Needs in die Menge `needs` aufgenommen. Die durch diese Instrumentierung gesammelten Daten können dann z. B. mit Hilfe eines Histogramms dargestellt wer-

den, um ein Gefühl für deren Häufigkeit zu erhalten. In Abb. 9-14 haben wir dies getan.

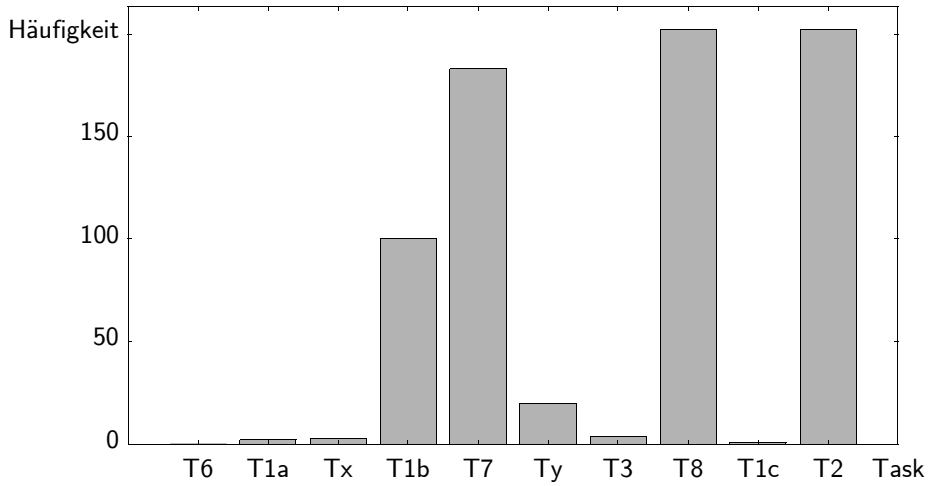


Abbildung 9-14. Histogramm (Häufigkeit einzelnen Tasks in einem Testlauf)

Die Abbildung zeigt die Ergebnisse eines Laufs des *RoomAutomation*-Beispiels, bei welchem alle Sensoren durch randomisierte Stubs ersetzt wurden. Die Tasks dieser Sensoren wurden aus der Betrachtung herausgenommen, da die Stubs nicht Teil des endgültigen Systems sind. Die Resultate wurden auf der Basis von Messungen einer alternativen Spezifikation des *RoomAutomation*-Beispiels gewonnen, welches in der Fallstudie *RoomAutomationX* (siehe Abschnitt 10.2.2 auf Seite 316) erstellt wurde. Daher entsprechen die Task-Namen nicht exakt denen von *RoomAutomation* (insbesondere wurde eine Verfeinerung von T1 in T1a, T1b und T1c vorgenommen und zwei neue Tasks Tx und Ty eingeführt). Die Häufigkeiten wurden innerhalb eines Testlaufs von 1000 Sekunden ermittelt.

Man sieht deutlich, dass Task T6, welcher die Blendung vermeiden soll, in diesem Testlauf nie überprüft wurde. Dies kann daran liegen, dass der Benutzer keine Blendungssituation evaluiert hat oder keine solche Situation evaluieren konnte (weil sich der Raum z.B. auf der Schattenseite des Gebäudes befindet). Ein weiterer Grund für die obige Beobachtung kann aber auch eine fehlerhafte Realisierung des *RoomAutomation*-Systems (insbesondere des GlareCtrl-Objekttyps) sein.

9.3.3 Vollautomatisches Prototyping („virtuelles Labor“)

Bisher musste der Entwickler (Modellierer oder Tester) ausgehend vom Urteil eines Tests (siehe Abb. 9-12 auf Seite 277) oder den Ergebnissen der dynamischen Analyse (siehe oben) neue Testfälle erzeugen und bei Bedarf das Testobjekt (bzw. den Prototyp oder dessen Spezifikation) modifizieren.

Wenn es gelänge diese Aktivitäten der Test- und Analyseplanung zu automatisieren, so könnte man damit einen vollständig automatisierten Entwicklungszyklus realisieren. In Abb. 9-15 ist eine Skizze eines solchen vollständig virtualisierten Software-Entwicklungslabors („*virtuelles Labor*“) gezeigt.

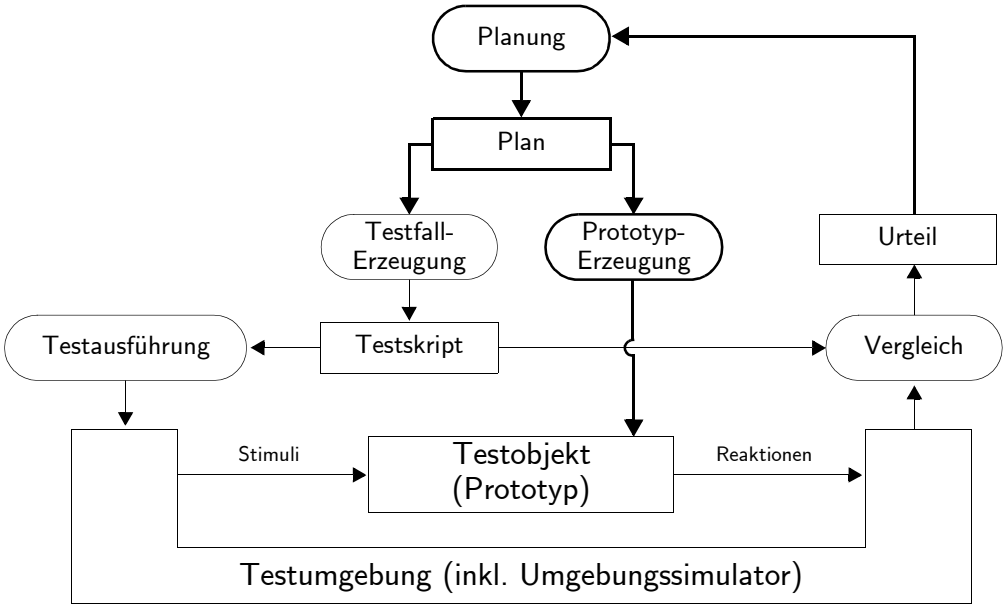


Abbildung 9-15. Erweiterung der Testumgebung zu einem „virtuellen Labor“

Diese Abbildung ist eine Erweiterung von Abb. 9-12 und zeigt neben der zusätzlichen Planungsebene auch die Prototyp-Erzeugung, also die Modifikation des Testobjekts (die Erweiterungen sind dick umrandet). Alle diese Aktivitäten müssen automatisch ausführbar sein, um einen vollautomatischen Gesamtablauf zu erreichen.

Der gesamte Prozesszyklus, wie er zur Realisierung des „virtuelles Labor“ im Kontext der PROBAnD-Methode nötig wäre, ist in Abb. 9-16 gezeigt, wobei die zusätzlich benötigten Elemente wiederum durch dicke Umrandungen gekennzeichnet sind.

Für die *Prototyperzeugung* stehen die in Abschnitt 9.2 auf Seite 258 eingeführten Werkzeuge zur Verfügung, mit welchen aus der Anforderungsspezifikation automatisch ein lauffähiger Prototyp erzeugt werden kann.

Während der *Prototypausführung* werden die Ein- und Ausgaben des Prototyps über die Testumgebung protokolliert und, wenn vorhanden, die Daten einer zusätzlichen Instrumentierung aufgezeichnet. Für diese Ausführung ist insbesondere die in Abschnitt 9.1.3 auf Seite 250 und in Abschnitt 9.2.3 auf Seite 268 erläuterte Verschmelzung von Prototyp und dessen Umgebung (Simulation) sinnvoll, da dadurch eine Beschleunigung dieser Ausführung und somit der Messdatenerfassung erfolgen kann.

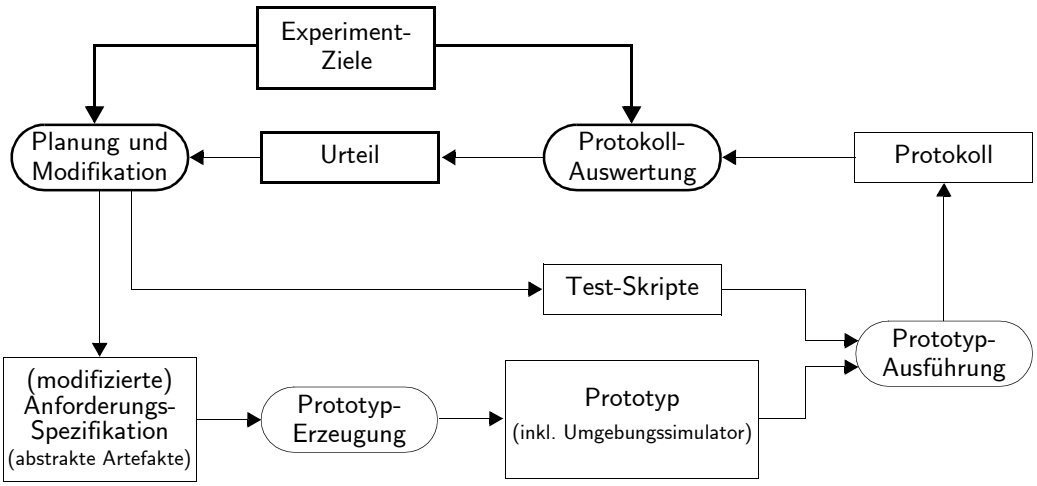


Abbildung 9-16. Prozessmodell des „virtuellen Labors“

Beginnend bei der *Protokollauswertung* wird der Prozesszyklus mit Hife zusätzlicher Werkzeuge weitergeführt. Diese wurden von Rehm im Rahmen einer Diplomarbeit [Reh04] erstellt, wobei die Aktivitäten des „virtuellen Labors“ so aufgeteilt wurden, dass ein sog. *Evaluator*-Tool die Protokollauswertung und ein *Modifier*-Tool die *Planung* neuer Testfälle und die *Modifikation* des Testobjekts bzw. der Test-Skripten übernimmt. Dabei erfolgt die Planung und die Beurteilung der Ergebnisse auf der Basis der gesteckten *Experimentziele*, welche spezifisch für das jeweils gewünschte Ziel der Entwicklung sind. Wir werden diese weiter unten an einem konkreten Beispiel illustrieren.

Neben den Werkzeugen zur Automatisierung der eigentlichen Entwicklungsaktivitäten musste ein weiteres Werkzeug implementiert werden, welches die Ausführung der einzelnen Werkzeuge koordiniert und steuert. Da es sich hierbei um einen einfachen zyklischen Aufruf der automatisierten Aktivitäten handelt, lässt sich dies einfach in einem Shell-Skript (z.B. unter UNIX realisieren). Rehm ist in seiner Arbeit einen Schritt weitergegangen und hat diese Steuerung um eine grafische Oberfläche erweitert, welche eine einfache Änderung und Beobachtung der einzelnen Prozessschritte ermöglicht.

Im Besonderen wurde in den Werkzeugen des „virtuellen Labors“ eine automatische Archivierung aller Protokolle, Test-Skripten und Anforderungsspezifikationen vorgenommen, um ein Experiment jederzeit nachvollziehen zu können.

Automatisierte Modifikation

Als generische Form der Modifikation von Spezifikationen lässt sich die Änderung von Attributbelegungen identifizieren, die für alle Modelle, die eine Instanz des PROBAnD-Produktmodells darstellen, identisch sind. Diese Modifikation wurde

von Rehm in das Modifier-Werkzeug aufgenommen und soll hier mit Hilfe von AL++ dargelegt werden.

Wir beginnen dabei mit der Definition der Operation `incrementAttribute()` in der RequirementsSpecification:

```
1   /* RequirementsSpecification: */
2   public boolean incrementAttribute(String cotName, String attrName,
3    Double upperLimit, Double increment) {
4       foreach(ControlObjectTypeConfiguration cotc in
5        itsControlObjectTypeConfiguration) {
6           if(cotc.name.equalsIgnoreCase(cotName)) {
7               return cotc.incrementAttribute(attrName, upperLimit,
8                increment);
9           }
10      }
11      return false;
12  }
```

Hier wird zunächst die relevante ControlObjectTypeConfiguration über einem Namensvergleich gesucht und die Aufgabe an die `incrementAttribute()`-Operation der gefundene Instanz weitergereicht:

```
1   /* ControlObjectType: */
2   public boolean incrementAttribute(String attrName, Double upperLimit,
3    Double increment) {
4       foreach(Attribute attr in itsAttribute) {
5           if(attr.name.equalsIgnoreCase(attrName) {
6               return attr.increment(upperLimit, increment)
7           }
8       }
9       return false;
10  }
```

Auch hier wird wieder über einen Vergleich von Namen das zu modifizierende Attribut gesucht und die eigentliche Änderung der Attributbelegung an die `increment()`-Operation des abstrakten Artefakttyps Attribute delegiert:

```
1   /* Attribute: */
2   public boolean increment(Double upperLimit, Double increment) {
3       Double attrValue = new Double(this.value);
4       if(attrValue+increment > upperLimit) {
5           return false;
6       }
7       attrValue = attrValue + increment;
8       this.value = attrValue.toString();
```

```
9         return true;
10    }
```

Wir gehen bei der Realisierung der Operation davon aus, dass nur zahlenwertige Attribute zu verändern sind. Daher können wir alle Attributwerte als `Double`-Zahlen darstellen. Nachdem die aktuelle Variablenbelegung festgestellt und die obere Grenze geprüft wurde, wird der Wert bei Bedarf erhöht und wieder in der Artefaktinstanz gespeichert. Eine solche erfolgreiche Modifikation wird mit `true` bestätigt.

Neben solchen generischen Modifikationen sieht das Modifier-Werkzeug auch die Integration experimentspezifischer Modifikationen (zum Beispiel das Hinzufügen eines Heizkörpers oder aber auch die Änderung von Attributwerten, die sich nicht wie oben behandeln lassen) vor. Diese können in der technischen Realisierung mittels Java-Reflection ohne eine Änderung des generischen Teils des Werkzeugs eingebunden werden (vgl. [Reh04, S.22f.]).

Automatisierte Planung

Als mögliche Planungsstrategie wurde eine Optimierung der Spezifikation ausgewählt und implementiert, da Rehm in seiner Arbeit die bestmögliche Parametrisierung eines komplexen Reglers untersuchte. Dazu erhält der Modifier eine Konfigurationsdatei mit den Experimentzielen als Eingabe, in welcher die gewünschten Modifikationen (Name des zu modifizierenden Attributs und des Control-Object-Types, untere und obere Grenze und Inkrement) aufgeführt sind. Die Planung der als Nächstes durchzuführenden Modifikation erfolgt dabei auf der Basis der *Leistung* (oder „*Performance*“) der Spezifikationen aller bisher durchgeführten Experimente. Die Berechnung der Performance erfolgt durch eine automatisierte Auswertung im Evaluator-Werkzeug (siehe unten).

Zu Beginn einer Experimentdurchführung werden alle Attributwerte auf die untere Grenze gesetzt und die Performance dieser Spezifikation berechnet. Die jeweils nächste Parametrisierung, die untersucht werden soll, wird dann ausgehend von der bisher als optimal identifizierten Anforderungsspezifikationen durchgeführt (siehe Abb. 9-16 auf Seite 287), indem einer der Attributwerte inkrementiert wird. Somit bewegt man sich stets entlang der lokalen Minima bis es keine weitere möglichen Änderungen der optimalen Spezifikation gibt. Die Suche wird dann bei den noch nicht vollständig veränderten Spezifikationen fortgeführt und man erhält am Ende – wenn man die automatische Durchführung nicht vorher beendet – eine vollständige Abdeckung aller Möglichkeiten. Diese Suche entlang der lokalen Minima macht Sinn, wenn das Ziel ein Unterschreiten einer vorher festgelegten Optimialitätsgrenze ist, weil man dann unter Umständen die Experimente schon vorher beenden kann und damit Zeit einspart.

Fallstudie: „Regelungsoptimierung“

Wie wir bereits erläuterten, ist die Bewertung von Spezifikationen auf der Basis von Messergebnissen an die Spezifika der jeweiligen Entwicklungsziele anzupassen. Wir werden zur Vorstellung einer möglichen Realisierung des Evaluator-Werkzeugs daher das Beispiel aus Abschnitt 9.3.1 aufgreifen.

Dort hatten wir bei der Illustration der Zufallstests einen Temperaturverlauf vorgestellt (Abb. 9-13 auf Seite 279), bei welchem die Belegung eines Raumes einen Einfluss auf dessen Temperaturregelung hatte. Wir vermuteten, dass die Größe der Verzögerung Δt_{occ} einen Einfluss auf die Energieeinsparung hat. Dies wollen wir nun mit Hilfe eines Experiments im „virtuellen Labor“ überprüfen und – wenn möglich – die Form dieses Zusammenhangs quantifizieren.

Zunächst definieren wir also als Experimentziel, die Zeiten Δt_{occ} zwischen 0s und 3900s (65min) in Schritten von 3,25 Minuten zu variieren. Als Performance einer Spezifikation fassen wir dabei deren Energieverbrauch auf. Da dieser Verbrauch proportional zur Wärmemenge Q ist, ist die Performance P umso besser, je geringer Q ist.

Das Ergebnis ist in Abb. 9-17 für verschiedene Anfangswerte r („random seed“) des Pseudo-Zufallsgenerators gezeigt.

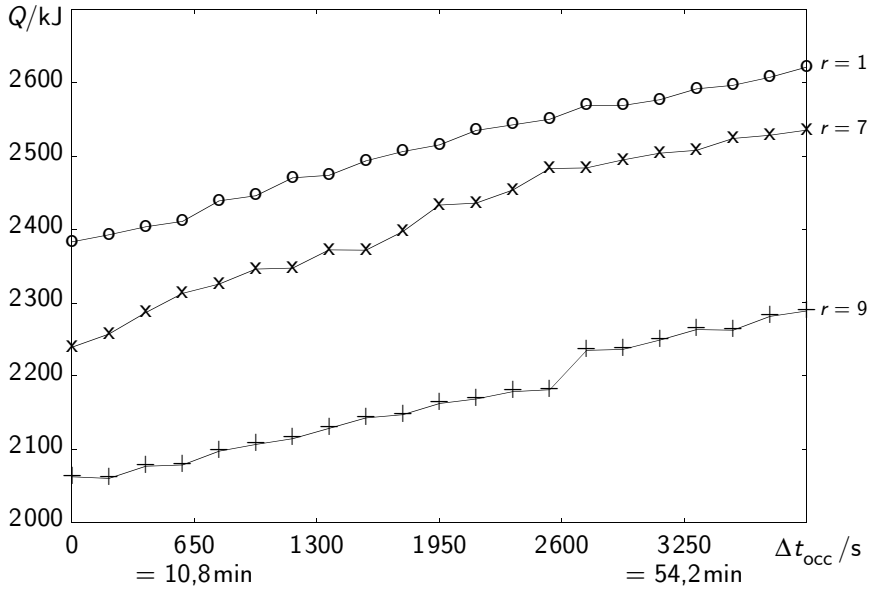


Abbildung 9-17. Energieverbrauch in Abhängigkeit von Verzögerungszeit

Wie man erkennt, zeigt diese Abbildung deutlich unterschiedliche Ergebnisse, je nachdem welcher Anfangswert für den Zufallsgenerator gewählt wurde. So entspricht der Punkt $(1950, 2482)$ für $r = 11$ z.B. dem Ergebnis des einzelnen Laufs aus Abbildung Abb. 9-13.

Um verlässlichere Aussagen zu erhalten, variierten wir jeweils Δt_{occ} für 30 verschiedene Werte von r ($r = 1, 3, 5, \ldots, 59$). Insgesamt ergibt sich dadurch eine Zahl von 630 Einzelexperimenten, die wir automatisch durchführten. Die Ergebnisse sind in Abb. 9-18 dargestellt.

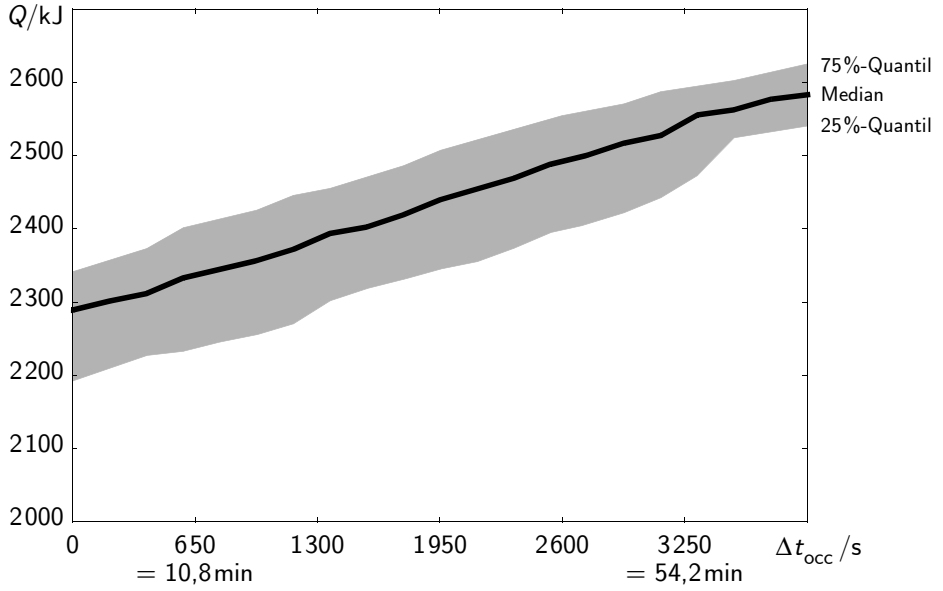


Abbildung 9-18. Mittlerer Energieverbrauch in Abhängigkeit von Verzögerungszeit

Die dicke Linie zeigt den Median aller Einzelergebnisse, der graue Bereich gibt die Streuung der Werte an. Die untere Grenze des grauen Bereichs ist das jeweilige 25%- und die obere Grenze das jeweilige 75%-Quantil (vgl. Abschnitt 10.2.1 auf Seite 310).

Die Abhängigkeit des Energieverbrauchs von Δt_{occ} ist darin deutlich zu erkennen. Mit der Berechnungsvorschrift, die in Abschnitt 10.2.1 vorgestellt wird, ergibt sich eine mittlere Steigung von 75 J/s, d.h. der Energieverbrauch steigt im Mittel um 15,8 kJ pro Zunahme der Verzögerung um 3,5 Minuten. Ein sofortiges Abschalten würde also die größte Einsparung bringen.

Der Energieverbrauch alleine reicht allerdings nicht aus, um ein wirklich leistungsfähiges MSR-System zu erhalten. Das optimale System wäre nämlich ein solches, in welchem nie die Solltemperatur erreicht wird sondern immer nur die abgesenkte Temperatur gehalten wird. Dass dies nicht den Kundenwünschen entspricht ist offensichtlich. Auch eine sehr kurze Einstellung von Δt_{occ} kann schlecht sein, da das sofortige Herabsetzung der Temperatur auch eine längere Aufwärmung des Raumes nach sich zieht und der Benutzer – wenn er zurückkehrt – somit zunächst einen kühleren Raum vor sich findet.

Es muss daher eine weitere Größe, der *Benutzerkomfort*, berücksichtigt werden (vgl. z.B. [PSS01, S.12]). Der Komfort hängt von Faktoren wie der Lufttemperatur, der Strahlungstemperatur der Wände, der Bekleidung, der körperlichen Aktivität

des Benutzers und der Strömungsgeschwindigkeit der Luft ab (siehe [ASH93, S.13.26f.]). Üblicherweise wird hier ein Formalismus nach Fanger [ASH93, S.8.18f.] eingesetzt. Bei dessen Berechnung sind komplexe Zusammenhänge (zum Teil auf der Basis experimentell gewonnener Kurven) zu berücksichtigen. Da dies den Rahmen dieser Arbeit sprengen würde, verwenden wir als eine einfachere Berechnungsvorschrift ein Maß, welches nur die Temperatur betrachtet.

Wir nehmen dazu an, dass alle anderen Größen, die einen Einfluss auf den Komfort haben, stets in einem annehmbaren Bereich sind und wählen das in [PSS01, S.103] vorgestellte TDL-Maß (Thermal Discomfort Level):

$$TDL = \frac{\vartheta_{\text{soll}} - \vartheta_{\text{ist}}}{\vartheta_{\text{soll}}} \cdot s_{\text{occ}} \qquad \textbf{(Gleichung 9-2)}$$

Dieses drückt den „Un“-Komfort durch die relative Abweichung der Isttemperatur von der Solltemperatur während der Raum belegt ist ($s_{\text{occ}} = 1$) aus.

Den „Un“-Komfort U eines gesamten Testlaufs berechnen wir als den Mittelwert des Quadrats von TDL, womit sich bei n Messwerten das Maß U ergibt zu

$$U = \left(\sum_{i=1}^{n} s_{\text{occ},i} \right)^{-1} \cdot \sum_{i=1}^{n} TDL_i^2 \qquad \textbf{(Gleichung 9-3)}$$

Wir wollen annehmen, dass jeder Wert von U kleiner als $5{,}625 \cdot 10^{-3}$ (also eine mittlere Abweichung der Temperatur von weniger als 1,5 Grad bei einer Solltemperatur von $20\,^{\circ}\text{C}$) für einen Benutzer akzeptabel ist. Eine solche Vorgehensweise wurde auch von Morris et al. zur Beurteilung unterschiedlicher Steuerungsstrategien angewendet (siehe [MTB94]).

Die Größe s_{occ} erhalten wir wieder durch eine entsprechende Instrumentierung der Simulatorseite. Da wir das Personenverhalten durch Zufall bestimmen, nahmen wir für dieses Beispiel an, dass die Wahrscheinlichkeit, dass der Bewegungsmelder MotionSens keine Belegung meldet obwohl jemand im Raum ist, $1/4$ sei.

Der Verlauf von U in Abhängigkeit von Δt_{occ} ist in Abb. 9-19 für die identischen Parameter r wie in Abb. 9-17 auf Seite 290 gezeigt. Auch hier erkennt man wieder eine Abhängigkeit von den zufälligen Bewegungsereignissen, die um einiges deutlicher zu Tage tritt als bei der Betrachtung des Energieverbrauchs. Insbesondere die Daten für $r = 9$ scheinen sich atypisch zu verhalten. Vergleicht man dies mit dem Verlauf des Energieverbrauchs, so stellt man fest, dass auch dieser niedriger ist als andere. Eine Untersuchung der Einzeldaten, also des zeitlichen Verlaufs der Messgrößen, liefert die Begründung (siehe Abb. 9-20).

Die Bewegungsereignisse für $r = 9$ sind so verteilt, dass der Raum häufig leer steht und daher öfter nur die niedrigere Sollvorgabe von 15 Grad eingehalten wer-

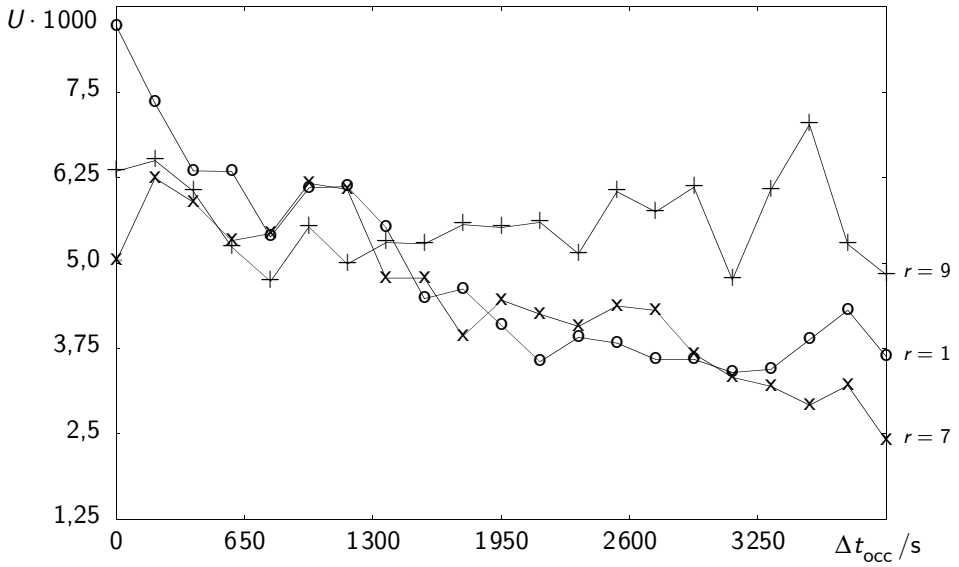


Abbildung 9-19. „Un"-Komfort in Abhängigkeit von Verzögerungszeit

den muss. Da die Perioden, in welchen sich eine Person im Raum befindet, nur sehr kurz sind, macht sich der höhere TDL zu Beginn einer Aufheizperiode stärker im Wert U bemerkbar.

Um mit einem mittleren Wert für U rechnen zu können, wurde wieder der Median und die 25%- und 75%-Quantile bestimmt. Die Resultate zeigt Abb. 9-21.

Man erkennt, dass U mit der Zunahme von Δt_{occ} abnimmt, d.h. je länger die Verzögerung wird, desto größer wird auch der vom Benutzer empfundene Komfort (im Mittel nimmt der Komfort pro Zunahme von Δt_{occ} um 3,5 Minuten um $0{,}12 \cdot 10^{-3}$ Punkte zu). Ein System mit höchstem Komfort wäre also ein solches mit $\Delta t_{\text{occ}} = \infty$.

Die Energieeinsparung und der Benutzerkomfort stehen damit in Konkurrenz zueinander. Interessant ist es also folglich, ein System so zu konzipieren, dass man eine Performance erreicht, bei welcher beide Aspekte möglichst optimal sind. Mit der

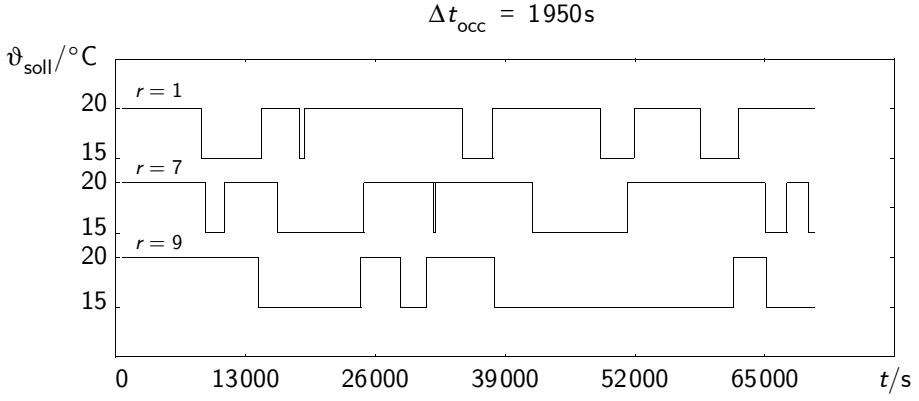


Abbildung 9-20. Vergleich der Sollvorgaben bei unterschiedlichen Zufallsereignissen

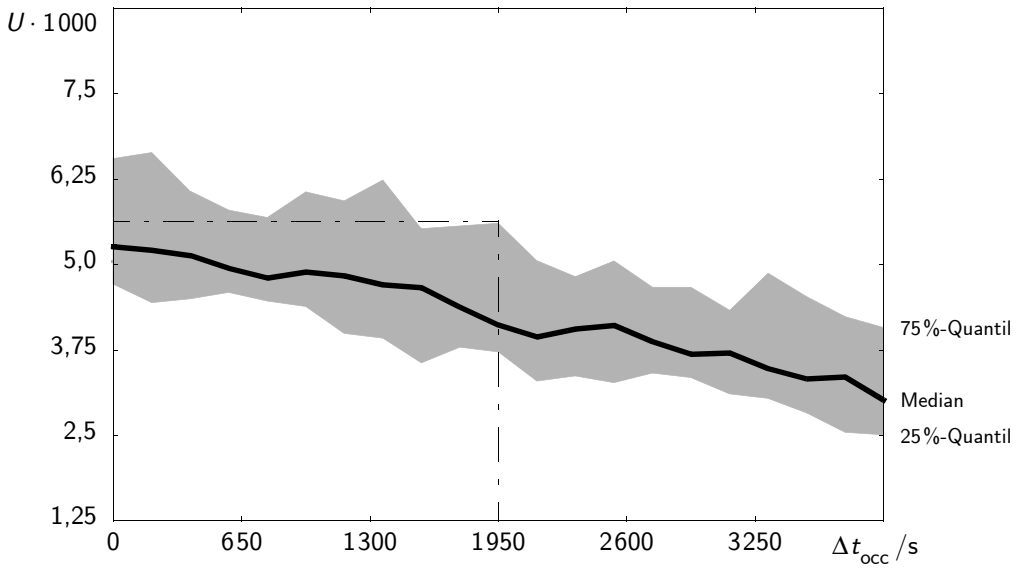


Abbildung 9-21. Mittlerer „Un"-Komfort in Abhängigkeit von Verzögerungszeit

obigen Voraussetzung von $U < 5{,}625 \cdot 10^{-3}$ für einen akzeptablen Komfort, liegt der optimale Wert von Δt_{occ} bei 1950 Sekunden (oder 32,5 Minuten), da hier auch der minimale Energieverbrauch zu finden ist (als Kriterium wählten wir ein Δt_{occ}, für welches U kleiner als der 75%-Quantil-Wert ist und damit 75% der Fälle erfasst werden; siehe Geraden in Abb. 9-21).

Eine Erweiterung des initialen *RoomAutomation*-Systems um die Strategie zur Reduktion der Raumtemperatur nach einer gewissen Zeit macht also durchaus Sinn, weil dadurch – ohne eine nennenswerte Einschränkung des Komforts – eine relevante Energieeinsparung erzielt werden kann. Für obiges Beispiel ergab sich bei dem gewählten Δt_{occ} eine mittlere Reduktion von Q um 260kJ (bei einem Betrachtungszeitraum von 21,6 Stunden), was einer Verringerung des Energieverbrauchs um fast 10% gegenüber der Regelung ohne eine Absenkung entspricht.

Die Frage, die sich nun stellt, ist, ob dieser Wert auch für andere Szenarien vernünftig ist, ob man Δt_{occ} also sinnvoll statisch festlegen kann. Der Vorteil wäre, dass die Regelungsimplementierung einfacher würde, da man keine dynamische Anpassung der Verzögerungszeit vornehmen müsste. Damit würde der Entwicklungsaufwand sinken (da weniger Features spezifiziert und getestet werden müssten) aber auch – was für den Bereich der eingebetteten System ein wichtiger Punkt ist – die Kosten des einzelnen Controller-Bauteils reduziert werden (es genügt evtl. ein schwächerer Prozessor und weniger Speicher).

Die Vermutung legt allerdings nahe, dass eine solche statische Festlegung nicht möglich ist, da wir Δt_{occ} für die gewählte Zufallsverteilung optimiert hatten. Wir können dies wieder durch Experimente in unserem „virtuellen Labor" untersuchen. Dazu betrachten wir zunächst, wie stark der längstmögliche Zeitabstand Δt_{mot} zwi-

schen zwei Bewegungsereignissen die gemessenen Werte beeinflusst (zur Erinnerung: die Bewegungsereignisse werden aus dem Intervall $[0, \Delta t_{\text{mot}}]$ gezogen). Wir wählen die Parameterbelegungen $\Delta t_{\text{mot}} = 1950, 3250, 4550, 5850, 7150$ und 9750 Sekunden und erhalten damit bereits 3150 Einzelexperimente.

In Abb. 9-22 sind die Verläufe der 75%-Quantile für diese Messreihen gezeigt.

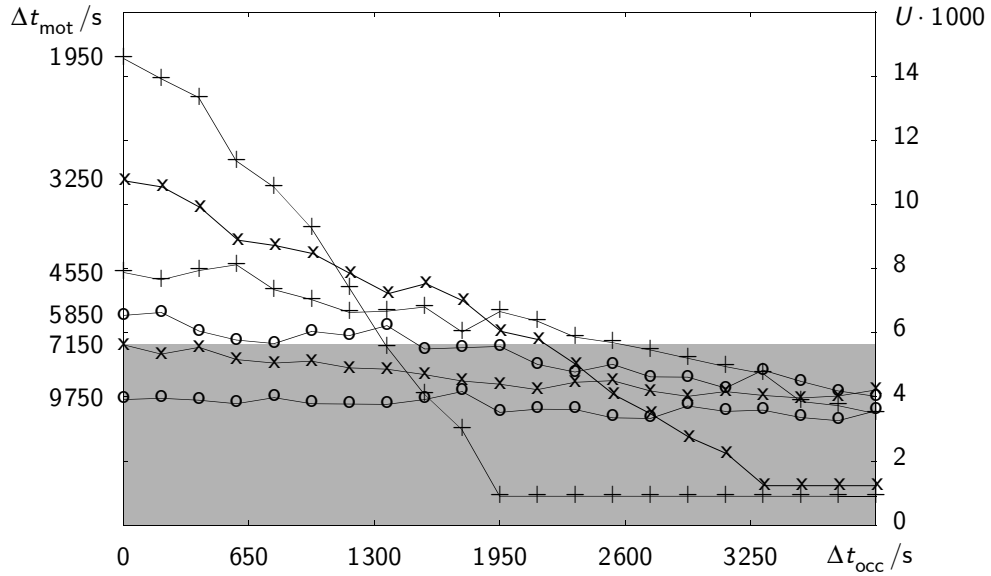


Abbildung 9-22. 75%-Quantile des „Un"-Komforts in Abhängigkeit von Belegungsdauer

Der graue Bereich kennzeichnet alle Werte von U, die akzeptabel sind. Der Energieverbrauch nahm – wie auch bei dem ersten Experiment – mit der Verlängerung von Δt_{occ} zu, weshalb jeweils der kleinste Wert von Δt_{occ}, für welchen die Schwelle zum grauen Bereich unterschritten wird, die optimale Parametrisierung darstellt.

Ein besonderes Augenmerk sollte man auf die Verläufe für $\Delta t_{\text{mot}} = 1950$s und für 3250s richten, da bei diesen Verläufen für ein $\Delta t_{\text{occ}} \geq \Delta t_{\text{mot}}$ keine Veränderung von U mehr zu beobachten ist. Dies liegt daran, dass es ab diesem Punkt so gut wie unmöglich ist, dass es zu einer Absenkung der Raumtemperatur kommt, da fast alle Bewegungsereignisse innerhalb von Δt_{occ} stattfinden (es besteht nur eine kleine Wahrscheinlichkeit, dass ein Ereignis außerhalb von Δt_{occ} auftritt, da bei jeder „Ziehung" eines Zeitpunkts auch zur Hälfte die 1 für Bewegung und die 0 für keine Bewegung gezogen wird). Man erreicht für alle $\Delta t_{\text{occ}} \geq \Delta t_{\text{mot}}$ also immer einen niedrigstmöglichen „Un"-Komfort U_{lim}. Dabei nimmt U_{lim} mit größer werdendem Δt_{mot} zu, da sich der Zeitpunkt des ersten Bewegungsereignisses (und damit der Regelung) nach hinten verschiebt. Dadurch kann der Raum initial stärker auskühlen, was U dann höher ausfallen lässt.

Was an obigen Kurven auffällt ist, dass ab einem gewissen Wert von Δt_{mot} der optimale Wert von Δt_{occ} nicht mehr größer, sondern wieder kleiner wird. Wir können diesen maximalen Wert von Δt_{occ} ($\Delta t_{\text{occ, ideal}}$) also nutzen, um einen robusten

statischen Parameter für unsere Regelung festzulegen. Damit hat dieses Experiment unsere obige Vermutung (zumindest zum Teil) widerlegt.

Es bleibt nun also dieses maximale $\Delta t_{\text{occ, ideal}}$ zu bestimmen. Dazu erstellen wir ein mathematisches Modell der obigen Zusammenhänge auf der Basis der gemessenen Daten. Dieses erlaubt uns die analytische Bestimmung von $\max(\Delta t_{\text{occ, ideal}})$.

Zur Vereinfachung dieses Modells approximieren wir die ermittelten Kurven für U durch Geraden. In den meisten Fällen ist diese Approximationsgerade „kleiner" als die gemessenen Kurven, weshalb die am Ende erhalten Ergebnisse eventuell ein wenig zu „optimistisch" sein werden. Bei Bedarf ließe sich dies aber durch eine präzisere mathematische Beschreibung der Zusammenhänge genauer berechnen.

Als Nächstes müssen wir jeweils den y-Achsenabschnitt $b_0(\Delta t_{\text{mot}})$ und die untere Grenze von U, $U_{\text{lim}}(\Delta t_{\text{mot}})$, bestimmen, um zwei Punkte unserer Approximationsgeraden zu erhalten, womit wir diese dann analytisch beschreiben können. Wie man bereits in Abb. 9-22 erkennt, besteht zwischen diesen Größen kein linearer Zusammenhang. Wir ermitteln daher zunächst weitere Messwerte, um eine Regressionsrechnung durchführen zu können. Dazu variieren wir Δt_{mot} und betrachten die Ergebnisse für $\Delta t_{\text{occ}} = 0$ (b_0) und 10010s (U_{lim}). Die Ergebnisse sind in Abb. 9-23 zusammen mit den ermittelten Regressionskurven gezeigt (für den Fall $\Delta t_{\text{occ}} = 0$ konnten wir zusätzlich auf Resultate aus Abb. 9-22 zurückgreifen).

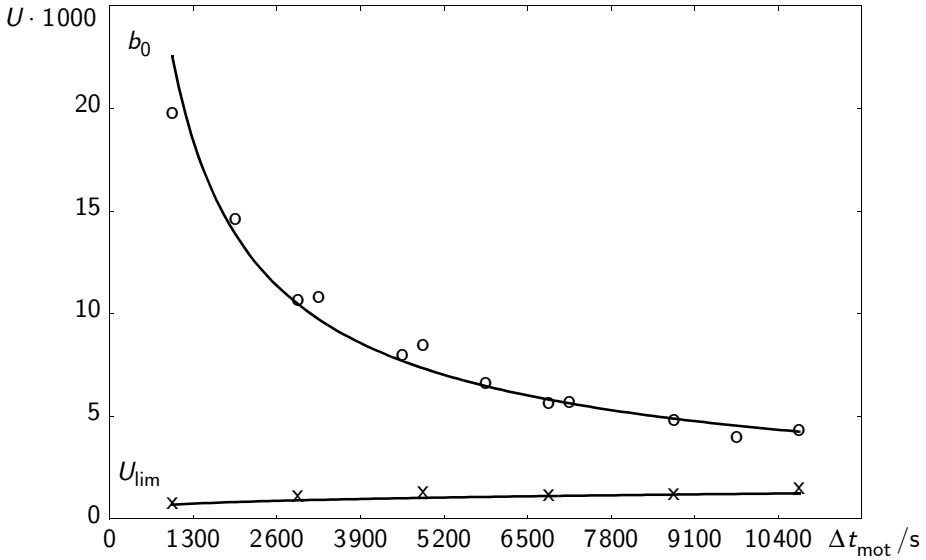


Abbildung 9-23. Regressionskurven zur Bestimmung der Approximationsgeraden

Es ergibt sich letztlich folgender Zusammenhang:

$$U(\Delta t_{\text{mot}}, \Delta t_{\text{occ}}) = \frac{U_{\text{lim}}(\Delta t_{\text{mot}}) - b_0(\Delta t_{\text{mot}})}{\Delta t_{\text{mot}}} \cdot \Delta t_{\text{occ}} + b_0(\Delta t_{\text{mot}}) \quad \textbf{(Gleichung 9-4)}$$

Berechnet man denjenigen Wert von Δt_{occ}, ab welchem $U(\Delta t_{\text{mot}}, \Delta t_{\text{occ}})$ unterhalb des zufriedenstellenden U-Werts von $5{,}625 \cdot 10^{-3}$ liegt, erhält man das jeweils ideale Δt_{occ} in Abhängigkeit von Δt_{mot}. In Abb. 9-24 ist $\Delta t_{\text{occ, ideal}}$ über Δt_{mot} aufgetragen.

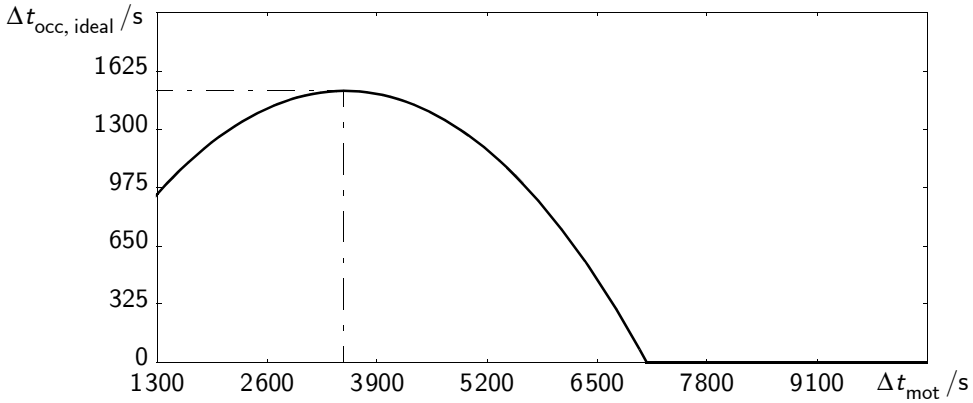


Abbildung 9-24. Verlauf der optimalen Verzögerungszeit

Hier erkennt man jetzt deutlich, dass ein Maximalwert eines optimalen Δt_{occ} existiert. Wir berechnen hierfür einen Wert von 1514s (= 25,2min). Dieser Wert ist allerdings geringer als der weiter oben für einen einzelnen Fall berechnete Wert. Dies liegt an der bereits erwähnten „Unterschätzung“ bei der linearen Approximation der U-Verläufe.

Wir wollen diesen Wert aber dennoch für eine statische Festlegung der Verzögerungszeit wählen (eine genauere Berechnung wäre wie erwähnt bei Bedarf möglich). Mit dieser Festlegung erzielen wir im Mittel eine Energieeinsparung von 296kJ oder eine Reduktion um ca. 11% gegenüber der primitiven Regelung.

Wir haben damit experimentell (in 4140 Einzelexperimenten) nachgewiesen, dass man durch eine sehr einfache Ergänzung einer Temperaturregelung eine relevante Energieeinsparung erreichen kann. Die Ergänzung erforderte lediglich die Spezifikation eines zusätzlichen Timers, eines weiteren Attributs und eines neuen Zustands (der notwendige Bewegungsmelder existierte bereits für die Helligkeitssteuerung).

Mit Hilfe des „virtuellen Labors“ konnte also eine begründbare Auswahl essenzieller Features des Systems vorgenommen werden. Damit erhält man einen systematischen Ansatz zur Vermeidung der Komplexität heutiger Software-Systeme, in welchen nur zu häufig „nice-to-have“-Features realisiert werden, da die Kunden sich bei der Auswahl der essenziellen Produktmerkmale schwer tun (vgl. auch [Wir95]).

Neben der Variation von Δt_{mot} müsste man nun allerdings noch die Robustheit des Reglers gegenüber anderen Einflüssen prüfen, so z.B. gegenüber der Außentemperatur, die wir bislang mit konstant 5°C angenommen hatten, oder einer veränderten Solltemperatur. Wir wollen es an dieser Stelle aber mit der Behandlung die-

ses Problems belassen und abschließend näher auf die Grenzen der hier vorgestellten Technik eingehen.

Diskussion und verwandte Arbeiten

Auch wenn man mit dem obigen „virtuellen Labor“ zahlreiche Experimente ohne großen Entwickleraufwand durchführen kann, stößt die „stupide“ Suche im Experimentraum schnell an ihre Grenzen, da die Anzahl der Einzelexperimente mit den möglichen Parameterkombinationen steigt. So hatten wir zur Untersuchung der Robustheit des Verzögerungsparameters Δt_{occ} bereits 3150 Experimente durchgeführt. Bei einer mittleren Dauer von 0,35 Minuten pro Gesamtzyklus des „virtuellen Labors“ (gemessen auf einem AMD Athlon PC mit 1,67GHz unter Windows XP) ergibt sich schon eine Wartezeit von 18,4 Stunden bis alle Ergebnisse vorliegen. Eine vernünftige Planung und Beobachtung der automatischen Experimentausführung durch einen menschlichen Experimentator ist also weiterhin sinnvoll.

Allerdings lassen sich die Experimente hervorragend verteilt zur Ausführung bringen. So muss nur nach jedem Einzelexperiment dessen Performance berechnet und an eine zentrale Stelle zur Planung des nächsten Experiments gemeldet werden. Führt man die Experimente ohne eine Suche entlang der lokalen Minima durch, dann muss man nur zu Beginn die jeweiligen Bereiche der Parameter auf die einzelnen Rechnerknoten verteilen. Man erhält damit einen skalierbaren „Algorithmus“, bei dem bei einer Verdoppelung der Problemgröße nur die Anzahl der Rechner verdoppelt werden muss, um das Ergebnis in jeweils gleicher Zeit zu erhalten. Die Arbeitsplatzrechner der Software-Entwickler böten sich z. B. für eine solche verteilte Rechnung an, die über Nacht läufen könnte.

Ein wichtiger Unterschied besteht zwischen den Experimenten in unserem „virtuellen Software-Labor“ und den automatisierten Experimenten, die heute in einem Labor der Naturwissenschaften möglich sind (vgl. z. B. den Nature-Artikel von King et al. [KWJ04]). In einem „Software-Labor“ hat man eigentlich alles Wissen zur Verfügung, um die gewünschten Ergebnisse auch analytisch abzuleiten. Da man insbesondere bei der Gebäudesimulation aber Differentialgleichungen erhält, die nur noch numerisch zu lösen sind, und bei Betrachtung von Personenverhalten auch (quasi-)zufällige Ereignisse relevant werden, ist eine solche formale Analyse vermutlich nicht praktikabel (siehe auch [ZiM04]). Auch viele Evaluierungen von Gebäuden basieren daher auf der Verwendung numerischer Simulatoren (vgl. [MTB94] und [PSS01]).

Im Bereich der Gebäudeautomation und -simulation gibt es viele Ansätze zur Untersuchung einer Kombination verschiedener Einflussfaktoren auf die Performance. So wird von Priolo et al. in [PSS01] der Beitrag selbstanpassender Algorithmen zur Verbesserung der Energieeffizienz von Gebäuden durch eine Simulation auf der Basis von Simulink (vgl. [Hof98]) untersucht. Morris et al. nutzen in [MTB94]

ebenfalls Simulatoren, um verschiedene Regelungsstrategien zu vergleichen (die Autoren geben auch einen Überblick über weitere Arbeiten auf diesem Gebiet). Rossi und Visioli untersuchen in [RoV95] den Einfluss der Raumgröße und -orientierung auf den Energieverbrauch. Gegenüber diesen Ansätzen zeigt sich die Stärke unseres Ansatzes in der direkten Ableitung des Testobjekts aus der Anforderungsspezifikation. Sobald eine Einigung über dessen Funktionalität (z.B. die Festlegung des Parameters Δt_{occ}) getroffen wurde, kann diese direkt in das endgültigen Produkt überführt (und wenn möglich sogar automatisch generiert) werden.

Dies trifft im Speziellen auch auf die auf den ersten Blick „umständlich“ erscheinende Modifikation der Attributbegelungen im Modell an Stelle deren Beschreibung in einem externen Konfigurations-File zu. Würde man aber dynamische Parameterbelegungen zulassen, so müsste man die Spezifikation (und damit das Testobjekt) um diese Funktionalität erweitern. Diese wäre aber nie Teil des endgültigen Produkts, was neben einer eventuellen Beeinflussung der Laufzeiteigenschaften auch implizierte, dass man ein anderes Produkt als das später ausgelieferte testen würde.

Schließlich kann mit dem vorgestellten Mechanismus jede PROBAnD-Spezifikation bzgl. beliebiger Parameter (Attributbelegungen) modifiziert werden, ohne dass eine manuelle Änderung der Spezifikation nötig wäre. Damit wird es für einen Tester ein Leichtes, das Testobjekt nach seinen Wünschen zu modifizieren, und damit eventuell auch Auswirkungen von Modifikation, die vom Modellierer nicht vorgesehen waren, zu überprüfen.

Den Preis, den man für eine solche Flexibilität zahlt, ist eine verlängerte Zeit für einen einzelnen Prozesszyklus, da die Modifikation der Spezifikation und die Generierung des Prototyps jedesmal aufs Neue durchgeführt werden müssen. Dieser Anteil beträgt für das obige Experiment 78,5% der Gesamtzykluszeit. Dabei entfallen 2,5% auf die Modifikation der abstrakten Artefakte (Modifier), 10% auf die Prüfung der Konsistenz und deren Wiederherstellung (Checker), 11% auf das Unparsen zur Erzeugung der SDL-Dokumente, 16% auf die C-Code-Generierung und schließlich 39% auf das Kompilieren. Die Ausführung des Prototyps benötigt lediglich 18% und die Evaluierung der Resultate 3,5% der Gesamtzeit eines Einzelexperiments.

Zusammenfassung

Nach einer einleitenden Einführung in die Grundlagen und möglichen Probleme eines Prototypings bot dieses Kapitel eine Darstellung möglicher Automatisierungsansätze. Als Probleme wurden die Unterschiede zwischen Spezifikation und Prototyp herausgearbeitet, die Relevanz der physikalischen Umgebung für ein reaktives System diskutiert und auf unterschiedliche Zeitbasen eingegangen.

Zu den behandelten Automatisierungsansätzen gehörten die automatische bzw. computer-unterstützte Selektion relevanter Eigenschaften aus einer Anforderungs-

spezifikation und die Generierung von Prototypen ausgehend von einer solchen veränderten Spezifikation. Daneben wurde gezeigt, wie ein Prototyp und dessen Umgebung (bzw. ein Umgebungssimulator) gekoppelt werden können. Insbesondere wurde auf die Variante der Verschmelzung zu einem geschlossenen System eingegangen. Für alle diese Aktivitäten wurde eine Realisierung mit AL++ vorgestellt, bzw. illustriert, wie eine solche Umsetzung aussehen würde.

Zum Abschluss wurde erläutert, wie das Testen mit einem Prototypen automatisiert werden kann, also der Prototyping-Schritt im weiteren Sinn. Dazu wurden existierende automatische Testtechniken aufgeführt und zum Ende des Kapitels das Konzept des „virtuellen Software-Labors“ eingeführt. In einem solchen „Labor“ konnten eine Vielzahl von Einzelexperimenten vollautomatisch durchgeführt werden und damit in einem Anwendungsbeispiel eine relevante Energieeinsparung in einer Temperaturregelung durch die geringfügige Ergänzung deren Spezifikation erzielt werden.

10 Beurteilung des Automatisierungspotenzials

The first rule of any technology used in a business is that automation applied to an efficient operation will magnify the efficiency. The second is that automation applied to an inefficient operation will magnify the inefficiency.

— Bill Gates (amerik. Software-Unternehmer, Gründer von „Microsoft“, *1955)

Zum Abschluss dieser Arbeit soll das Potenzial einer Automatisierung genauer untersucht werden. Dazu gehört eine Beurteilung des Effizienz- bzw. Produktivitätsgewinns durch den Einsatz unserer Automatisierungswerkzeuge, wozu zunächst geeignete Techniken zur Messung der relevanten Eigenschaften eingeführt werden. Damit können dann die Ergebnisse einzelner Fallstudien und Experimente verglichen werden.

Neben solchen „positiven“ Ergebnissen werden auch die möglichen Grenzen, Risiken und Fallen einer Automatisierung vorgestellt und diskutiert.

10.1 Messungen

Zur Beurteilung einer Prozessverbesserung – wie in unserem Fall z.B. durch Automatisierung – werden konkrete Daten sowohl über die Produkte als auch über den eigentlichen Prozess benötigt. Dazu werden *Maße* eingesetzt. Ein Maß ist eine Messvorschrift, welche eine Methode zur Erhebung von Daten zur Charakterisierung eines bestimmten Merkmals des Untersuchungsgegenstandes beschreibt. Ein Maß ist formal eine Abbildung zwischen der realen Welt und einer formalisierten Welt, wobei empirische Relationen in der formalen Welt erhalten bleiben [VIS04].

In der Informatik-Literatur wird für den Begriff „Maß“ oft der nicht zutreffende Begriff „Metrik“ verwendet (siehe dazu auch die Erläuterungen in [Lig02, S. 212f.] und [Fen91, S. 21f.]). Unter einer *Metrik* versteht man eine Funktion m, die auf Paaren von Elementen (x, y) definiert ist, so dass $m(x, y)$ den „Abstand“ zwischen x und y beschreibt. Insbesondere muss gelten $m(x, x) = 0$, $m(x, y) = m(y, x)$ und $m(x, z) \leq m(x, y) + m(y, z)$ für alle x, y und z [Fen91, S. 59].

Da sich jedes Maß auf das jeweilige Produktmodell des Untersuchungsgegenstands bezieht [VIS04], gibt es keine allgemeingültigen Maße, mit denen wir unsere gewünschte Eigenschaften direkt messen könnten. In den folgenden Abschnitten werden wir daher spezifische Maße vorstellen, welche für die Beurteilung des Automatisierungsgewinns im Rahmen der PROBAnD-Methode eingesetzt wurden. Dabei kann analog zur Differenzierung des Begriffs „Qualität“ in Kapitel 7 zwischen *Produktmaßen* und *Prozessmaßen* unterschieden werden, wobei eine solche Klassifikation nicht unbedingt eindeutig zu treffen sein muss [May90, S. 555f.]. Dies liegt zum Teil darin begründet, dass – wie wir in Abschnitt 7.1 auf Seite 175 bereits festgestellt hatten – die Prozessqualität die Produktqualität beeinflusst und umgekehrt die Produktqualität von der Prozessqualität abhängt.

10.1.1 Messung von Produkteigenschaften

Produktmaße beziehen sich auf Produkteigenschaften und lassen sich daher durch eine direkte Betrachtung der Entwicklungsartefakte bestimmen. Wo es möglich ist, werden wir dabei im Folgenden *objektive Maße* [RoB87, S. 148] vorstellen, also solche, welche ohne eine subjektive Einschätzung durch die beteiligten Personen berechnet werden können. Nur solche Maße ermöglichen eine Reproduzierbarkeit von Resultaten [Lig02, S. 214f.] und dadurch letztlich einen wiederholbaren Vergleich der Ergebnisse. Dies ist wichtig, da in den von uns bearbeitetem Fallstudien jeweils andere Entwickler beteiligt waren.

Messung der Produktgröße

Die Größe der Artefakte ist eine der wichtigsten Produkteigenschaften, da diese i. d. R. mit den Kosten (Aufwand) und der Anzahl der Fehler zusammenhängt [May90, S. 580]. Auch für unsere Betrachtungen liefert die Berechnung der Größe eine wichtige Kennzahl, die es uns erst erlaubt die Ergebnisse unterschiedlicher Entwicklungsprojekte miteinander vergleichen zu können (siehe auch [Que02, S. 249ff.]).

Die wohl bekannteste und auch direkteste Bestimmung der Produktgröße erfolgt durch das Zählen der Code-Zeilen. Ganz unproblematisch ist dieses Maß verständlicherweise nicht, da nicht klar definiert ist, was eine Code-Zeile überhaupt ist. So ist es u. a. unklar, ob Kommentare beinhaltet sind, wie der Programmierstil ist, wel-

che Programmiersprache gewählt wurde, usw. Eine Diskussion weiterer Probleme dieses Maßes ist z.B. in [May90, S.581] und [GJM91, S.423f.] zu finden.

Eine Verbesserung gegenüber dem reinen Zählen der „*Lines of Code*“ (*LOC*) stellt das Messen der sog. „*Non-Commenting Source Statements*“ (*NCSS*) dar, welche keine Kommentare beinhalten und wobei auch Zeilenumbrüche korrekt verrechnet werden (für Details siehe z.B. [Gra92, S.233]).

Wie die Begriffe „Code“ und „Source“ schon andeuten, sind diese Maße für das Messen der Größe von Programmen vorgesehen und nicht so sehr für die Beurteilung der Größe anderer Artefakte wie z.B. grafischer Modelle. Auch in dieser Arbeit werden wir das NCSS-Maß nur für die Untersuchung der Automatisierungswerkzeuge einsetzen (siehe Abschnitt 10.2.3 auf Seite 329). Zur Beurteilung der in einem Entwicklungsprojekt erstellten Artefakte (z.B. die HTML- und SDL-Dokumente der PROBAnD-Methode, siehe Kapitel 6) führen wir ein eigenes Maß ein.

Da wir für alle unsere Typen von Entwicklungsdokumenten eine textuelle Darstellung erzeugen können (insbesondere SDL-PR für die SDL-Dokumente, siehe Abschnitt 2.2.2 auf Seite 19), bietet sich eine Variation des „Lines of Code“-Maßes an. Eine solche Variation ist deshalb nötig, weil die Dokumente keine Programmausdrücke beinhalten und die Dokumente auf der Basis von „Templates“ (Vorlagen) erstellt wurden. In Abschnitt 6.1.3 auf Seite 138 und insbesondere in dem Artikel [MeQ03] finden sich nähere Details zum Einsatz solcher Vorlagen.

Ein „faires“ Maß muss den Wiederverwendungsgewinn, der bei einem Einsatz von Vorlagen erzielt wird, herausrechnen. Daher gehen wir bei unserer Messmethode zweischrittig vor. Im ersten Schritt wird die Differenz zwischen Entwicklungsdokument und dessen Vorlage berechnet (mittels des UNIX-Werkzeugs `diff` [Dre93, S.372f.]). Dann wird an Hand dieser Differenz die Anzahl der hinzugefügten, geänderten und gelöschten Zeilen berechnet. Alle diese Zahlen gehen positiv in unser Maß ein. Insbesondere auch die gelöschten Zeilen, da ein Entfernen von Teilen aus einer Vorlage natürlich auch Entwicklungsaufwand bedeutet und daher mit der Produktgröße in Zusammenhang stehen muss.

Abbildung 10-1 zeigt ein Beispiel dafür, wie sich die Ausprägung dieses Maßes, wir wollen es im weiteren Verlauf *DDL* (für „*Differing Document Lines*“) nennen, für einen Ausschnitt aus einem SDL-PR-Dokument berechnet.

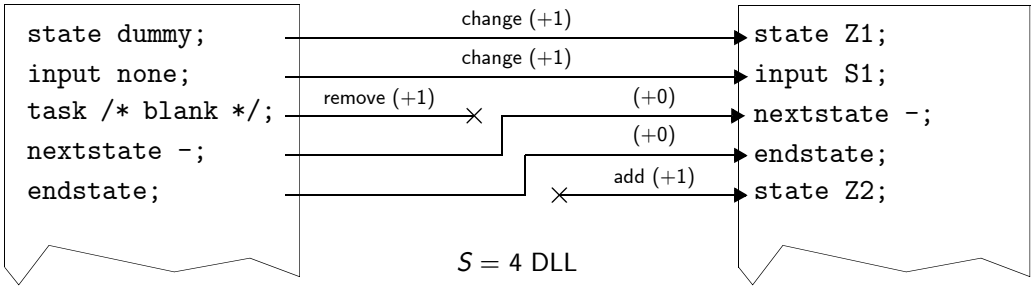


Abbildung 10-1. Berechnung der „Differing Document Lines“ (DDL) am Beispiel

Messung der Komplexität

Neben der reinen Größe eines Artefakts ist auch dessen interne Komplexität eine wichtige Eigenschaft, da diese Komplexität gewissermaßen die Schwierigkeit der Erstellung des Produkts widerspiegelt. Daher findet man auch hier – wie bei der Größe – einen Zusammenhang mit dem Aufwand und der Fehlerrate. Neben dieser internen (oder auch „psychologischen“ [Ebe98]) Komplexität existiert auch eine externe (oder algorithmische) Komplexität (siehe Abschnitt 7.1.1 auf Seite 177), die wir hier nicht näher betrachten werden.

Queins führt in seiner Arbeit [Que02, S. 250ff.] für die Berechnung der Komplexität ein Maß ein, bei welchem ausgehend von der Bewertung der „Komplexität“ einzelnen atomarer Artefakte durch die Entwickler, die Gesamtkomplexität der Dokumente hierarchisch berechnet wird. Da wir, wie oben bereits erwähnt, nicht auf dieselben Entwickler wie in den von Queins bearbeiteten Fallstudien (*FloorAutomation* und *FloorAutomationX*, siehe Abschnitt 8.2.2 auf Seite 224) zurückgreifen können, werden wir für die Komplexitätsberechnung ein objektives Maß einsetzen.

Für die Bestimmung der Komplexität werden in der Literatur verschiedene Maße vorgeschlagen, welche die funktionale Komplexität, die Datenkomplexität oder beide Aspekte messen (siehe [May90, S. 583ff.]).

Da unsere Entwicklungsprodukte stark funktionsorientiert sind (wir beschreiben das Verhalten eines reaktiven Systems), sind für unsere Anwendung die Maße der ersten Klasse geeignet. Eines der bekanntesten dieser Maße ist die von McCabe vorgeschlagene *zyklomatische Komplexität*, welche auf der zyklomatischen Zahl eines gerichteten Graphen G basiert [Jal97, S. 348ff.]. Diese Zahl beschreibt die Anzahl der möglichen Pfade durch den Graphen G. Dementsprechend berechnet sich die zyklomatische Komplexität C zu

$$C = |E| - |V| + 2\,, \qquad \textbf{(Gleichung 10-1)}$$

wobei E die Menge der Kanten (Edges) und V die Menge der Ecken (Vertices) ist (siehe auch Definition 2-4 auf Seite 15).

Ursprünglich für Kontrollflussgraphen in Programmen vorgeschlagen, lässt sich dieses Maß auch auf Entwicklungsdokumente übertragen, welche sich in geeigneter Weise als gerichteter Graph darstellen lassen [May90, S. 583f.]. Da die Funktionsbeschreibung in der PROBAnD-Methode über die Spezifikation von Zustandsautomaten erfolgt (welche durch gerichtete Graphen dargestellt werden können), lässt sich das McCabe-Maß hier ansetzen.

Eine zusätzliche Schwierigkeit bei der Berechnung von C für SDL-Zustandsgraphen ist die Möglichkeit in einem SDL-Zustandssymbol mehr als einen Zustandsbezeichner anzugeben oder mit dem „Asterisk“-Zustand („*“) alle Zustände zu referenzieren (siehe Abschnitt 6.2.3 auf Seite 151). Für die Berechnung von C

interpretieren wir daher diese Syntaxkonstrukte als abkürzende Schreibweisen und zählen die „echten“ Transitionen. Eine solche Auflösung des „*“-Zustandes hatten wir bereits bei der Visualisierung von Zustandsautomaten in Abschnitt 8.4.2 auf Seite 233 vorgenommen. Abb. 10-2 zeigt ein Beispiel für die Berechnung von C an Hand eines solchen SDL-Zustandsautomaten.

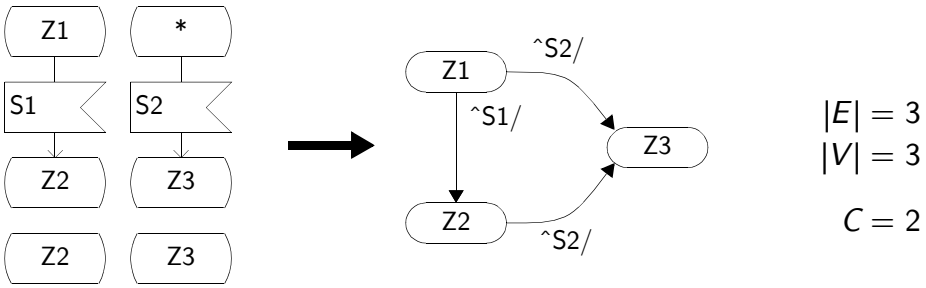


Abbildung 10-2. Berechnung der zyklomatischen Komplexität am Beispiel

Messung der Korrektheit und Zuverlässigkeit

Neben den „Ausdehnungs“-Merkmalen Größe und Komplexität ist für eine vollständige Beurteilung eines Produkts auch dessen Korrektheit (Übereinstimmung mit der Spezifikation) und Zuverlässigkeit (Erfüllung der Nutzervorstellungen) wichtig. Insbesondere bei der Betrachtung der Produktivität (oder Effizienz) eines Prozesses oder einer Methode genügt es nicht, nur die Größen der erstellten Produkte in Relation zum Entwicklungsaufwand zu setzen (siehe unten), sondern auch die Güte der Produkte muss in eine solche Bewertung einfließen.

Um die Korrektheit oder Zuverlässigkeit eines Produkts zu bestimmen, müsste man nun alle Fehler in einem solchen Artefakt messtechnisch bestimmen können. Dass dies nicht vollständig möglich ist leuchtet ein, da zum einen durch Testen nie alle Fehler gefunden werden können und zum anderen formale Verifikationstechniken oft nicht für sehr große Systeme anwendbar sind (vgl. auch Abschnitt 9.3.1 auf Seite 276). Desweiteren können Validierungstätigkeiten nicht vollständig automatisiert werden, da sie nur (manuell) von den Benutzern durchgeführt werden können. Auch Liggesmeyer weist in [Lig02, S. 231] auf diesen Sachverhalt hin und schlägt als Lösung vor, solche Größen auf der Basis von Erfahrungswissen aus messbaren Größen abzuleiten. Wir werden eine solche Möglichkeit bei der Diskussion der Prozessmaße weiter unten vorstellen.

In unserem Ansatz besteht allerdings die Möglichkeit, zumindest einen Teil der Fehler in den Artefakten direkt und automatisiert zu bestimmen. Eine Voraussetzung für ein korrektes Artefakt ist, dass dies vollständig und konsistent ist (vgl. Einfluss der internen auf die externe Qualität aus Abschnitt 7.1 auf Seite 175). In Abschnitt 8.1 auf Seite 188 hatten wir erläutert, wie Inkonsistenzen in den Dokumenten und den abstrakten Artefakten detektiert werden können. Zur Verein-

fachung der folgenden Abschnitte wollen wir dieses Prinzip nur für Inkonsistenzen in den konkreten Dokumenten betrachten. Damit können wir auf der Basis des Parser-Werkzeugs, welches die Verletzung der statischen Semantik in den konkreten Dokumenten meldet, die Anzahl der Inkonsistenzverletzungen in jedem Dokument zählen und erhalten damit die Zahl I.

Eine solche absolute Zahl I sagt direkt noch nichts über die Korrektheit des Artefakts aus. Wir müssen diese Zahl daher in Beziehung zu der Zahl der theoretisch maximal möglichen „Fehler“ $I_{\max}$ setzen und definieren damit einen *Konsistenzquotienten* κ wie folgt:

$$\kappa = 1 - \frac{I}{I_{\max}} \qquad \textbf{(Gleichung 10-2)}$$

Besitzt ein Artefakt also keine Inkonsistenz, so ist $\kappa = 1$ und falls die tatsächliche Fehlerzahl der maximalen Fehlerzahl entspricht ist der Quotient entsprechend $\kappa = 0$. Üblicherweise ist die Korrektheit eine binäre Größe (entweder weist das Artefakt keine Fehler auf und ist korrekt oder eben nicht korrekt, vgl. [Lig02, S.8]). Wir sind allerdings der Meinung, dass es insbesondere bei den Konsistenzverletzungen durchaus Sinn macht, auch Konsistenzquotienten aus dem Bereich $]0,1[$ zuzulassen. Vor allem bei der Anwendung von κ in Abschnitt 10.2 auf Seite 309 wird sich dies zeigen.

Was nun noch bleibt ist die Bestimmung von $I_{\max}$. Dazu nehmen wir an, dass ein Dokument, in welchem jeder „sinnvolle“ Text durch einen „sinnlosen“ Text ersetzt wird, eine solche maximale Fehlerzahl aufweist. Völlig sinnlos bedeutet dabei, dass der entsprechende Eintrag im Dokument ebenso durch Zufall hätte entstehen können und damit keinerlei Entwicklungsinformation trägt. Durch eine Modifikation unseres Unparsers können wir solche Dokumente leicht erzeugen. Dabei wird an Stelle der konkreten Repräsentation eines abstrakten Artefaktes jeweils ein zufälliger Text erzeugt. Mit dem Parser bestimmt man dann wieder die Zahl der Inkonsistenzverletzungen in solchen Dokumenten und erhält damit $I_{\max}$.

Die Auswahl der Zeichen und die Länge der erzeugten Texte erfolgte in unserer Realisierung dabei gleichverteilt aus einem jeweils vorgegebenen Intervall. Tabelle 10-1 zeigt die mit dieser Vorgehensweise randomisierte Variante der Tabelle 6-7 auf Seite 143 (ein „X“ bei Usage liefert stets eine Konsistenzverletzung).

Tabelle 10-1. Randomisierte Definition der Signaltypen (TempCtrl)

SignalType	*FormalParameter*	*Tasks*	*Usage*	*Description*
TOgQdaIW	Un, TQajVEIHvCEdPxx	XCYn, XrPw	X	HUtFbZjlnKBrNVMRhEtz
hRKBGhUwk	guV	IFI	X	WXiAKtKBekHioYaFY
ZYcQLxzQxKoXI	Df, CXZ	bqFQgMZo	X	kgrepkQGIjjFc

10.1.2 Messung von Prozesseigenschaften

Prozesseigenschaften werden während des Prozessablaufs, also während der *Erzeugung* der Produkte, gemessen.

Erfassung des Aufwands

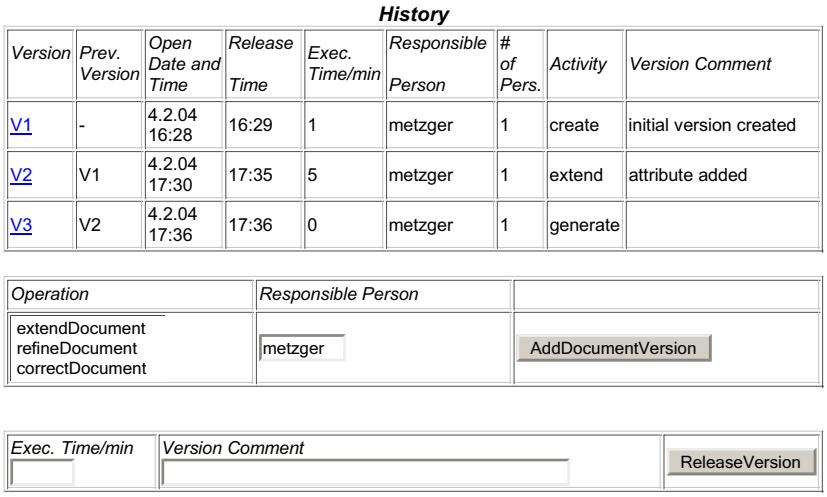
Die Prozesseigenschaft, die sich direkt in den Kosten der Erstellung eines Software-Produkts niederschlägt, ist der zeitliche *Aufwand E* (engl. „effort“). Will man genaue Untersuchungen der Effizienz des gesamten Entwicklungsprozesses vornehmen, so reicht die Betrachtung des Gesamtaufwands allerdings nicht aus, da auch die Verteilung des Aufwandes auf Entwicklungsphasen (bzw. Workflows) oder auf unterschiedliche Entwicklungsaktivitäten relevante Einblicke liefern kann. Insbesondere Leerlaufzeiten, die z.B. auf Grund einer starren Prozessdurchführung auftreten, können zu hohen Kosten führen. Queins hat in seiner Arbeit von diesen Auswertungsmöglichkeiten ausgiebig Gebrauch gemacht (siehe z.B. [Que02, S.254ff. und S.294ff.]).

Für unsere Betrachtungen ist speziell der pro Dokument angefallene Aufwand relevant, da wir diesen in Relation zur Dokumentgröße, bzw. -komplexität setzen werden. Weiter kann auch eine Zuordnung des Aufwands zu den einzelnen Klassen von Entwicklungsaktivitäten vorgenommen werden. Wie wir bereits in Abschnitt 2.2.2 auf Seite 19 aufgeführt hatten, existieren die von Queins identifizierten Klassen „*create*“, „*extend*“, „*refine*“, „*change*“, „*correct*“, „*test*“ und „*review*“. Um diese typischerweise manuellen Tätigkeiten von automatisierten Aktivitäten unterscheiden zu können, führen wir zusätzlich den Aktivitätstyp „*generate*“ ein.

Für die Aufwandserfassung steht uns ein rudimentäres Versionsverwaltungswerkzeug zur Verfügung, das einfache Versionierungsaufgaben über HTML-Formulare und CGI-Skripten realisiert (siehe [QuZ99, S.35]). Dieses Werkzeug wird seit längerem für unsere Arbeiten eingesetzt und bietet gegenüber mächtigen Tools wie CVS (siehe dazu nochmals Abschnitt 5.1 auf Seite 82) den Vorteil, dass eine präzise Zeiterfassung unterstützt wird. In Abb. 10-3 ist zur Illustration ein Auszug aus einer mit diesem Werkzeug erstellten Versionshistorie gezeigt.

Erfassung der korrigierten Fehler

Durch die Erfassung der „correct“-Aktivitäten mit Hilfe obiger Versionsverwaltung bietet sich eine recht einfach Möglichkeit, die Anzahl der korrigierten Fehler festzustellen (man zähle die Anzahl dieser Aktivitäten). Es ist dabei allerdings zu berücksichtigen, dass ein Fehler auch durch mehrere „correct“-Aktivitäten verbessert werden kann und mehrere Fehler durch nur eine solche Aktivität korrigiert werden



History

Version	Prev. Version	Open Date and Time	Release Time	Exec. Time/min	Responsible Person	# of Pers.	Activity	Version Comment
V1	-	4.2.04 16:28	16:29	1	metzger	1	create	initial version created
V2	V1	4.2.04 17:30	17:35	5	metzger	1	extend	attribute added
V3	V2	4.2.04 17:36	17:36	0	metzger	1	generate	

Abbildung 10-3. Beispielhafte Versionshistorie eines Entwicklungsdokuments

können. Es ist daher prinzipiell notwendig, die Fehler und Änderungen getrennt zu erfassen.

Den Aufwand einer solchen feingranularen Fehlererfassung und deren anschließende Auswertung hielten wir in unserer Arbeit allerdings für nicht gerechtfertigt, da einem die Zahl der *gefundenen* Fehler bei dem Vergleich zweier Projekte nicht unbedingt weiterhilft. Wie wir weiter oben schon andeuteten, sagt die Anzahl an gefundenen Fehlern nicht direkt etwas über die Korrektheit eines Artefakts aus, da die Gesamtfehlerzahl unbekannt ist.

Man könnte natürlich analog zu der Bestimmung des Konsistenzquotienten κ aus Gl. 10-2 auf Seite 306 versuchen, den *Korrektheitsgrad* k wie folgt zu bestimmen:

$$k = 1 - \frac{F}{F_{\max}} \qquad \textbf{(Gleichung 10-3)}$$

Dabei stellt $F_{\max}$ die Anzahl der tatsächlich enthaltenen (also maximal zu findenden) Fehler dar. Diese Größe lässt sich aus bereits dargelegten Gründen nicht direkt bestimmen. Man könnte aber eine Abschätzung von $F_{\max}$ auf der Basis von Erfahrungswissen vornehmen. Dabei kann man sich auf die Annahme stützen, dass die Anzahl der Fehler in einem Artefakt mit dessen Komplexität bzw. Größe korreliert, d.h. dass bei einer Zunahme der Komplexität bzw. Größe auch die Anzahl der fehler steigt. Damit könnte man dann bei bekannter Komplexität C die Größe $F_{\max}$ z.B. über eine lineare Funktion $F_{\max} = b_0 + b_1 \cdot S$ abschätzen. Die Koeffizienten b_0 und b_1 ließen sich durch eine Regression auf der Basis von Daten vergangener Projekte bestimmen.

Ein allgemeines Problem auch bei Einsatz solchen Erfahrungswissens ist, dass auch diese Daten nicht die maximale Fehlerzahl $F_{\max}$ sondern nur die Zahl an ge-

fundenen Fehlern, also F beinhalten. Nehmen wir aber für die folgenden Betrachtungen einmal an, dass F hinreichend nahe bei $F_{\max}$ läge.

Die gesammelten Daten für die Fallstudie *FloorAutomationX* (siehe unten), welche die Referenzwerte für die Beurteilung des Automatisierungspotenzials darstellen, sind in Abb. 10-4 als Streuungsdiagramm dargestellt.

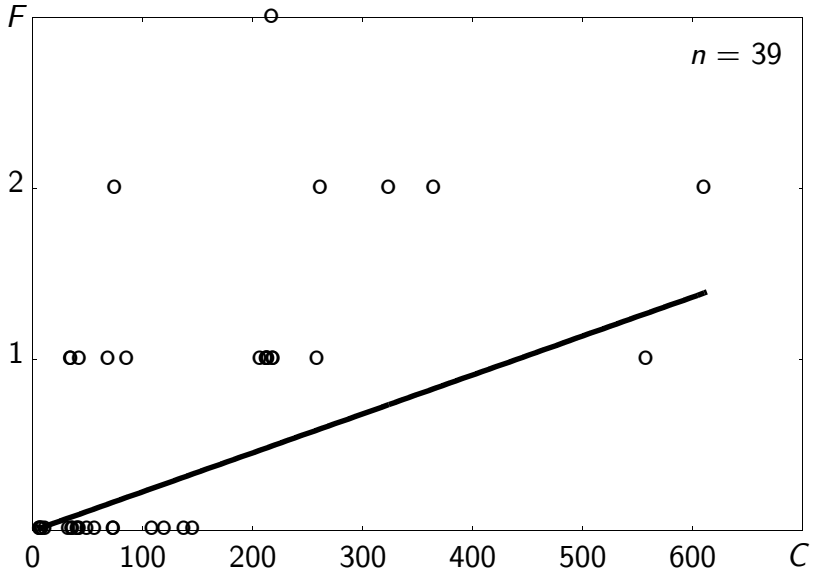


Abbildung 10-4. Streuungsdiagramm für SDL-Dokumente von *FloorAutomationX*

Obwohl hier scheinbar eine Korrelation zwischen C und F besteht (die komplexeren Artefakte haben etwas mehr Fehler), zeigt die Regressionsgerade zum Teil deutliche Abweichungen von den tatsächlichen Messwerten. Desweiteren stellt die geringe Zahl an Fehlern pro Artefakt (Maximum ist 3) und die Ganzzahleigenschaft der Fehler eine weitere Ungenauigkeit dar (die Punkte auf der Regressionsgeraden sind reellwertig). Auf Grund dieses Datenmaterials können wir für unsere Betrachtungen (auch mit der getroffenen Annahme) nicht von der obigen Abschätzung Gebrauch machen.

Wir werden daher – so weit es möglich ist –, die in Abschnitt 10.1.1 vorgestellten Produktmaße als „vertrauenswürdigere“ Maße im weiteren Verlauf einsetzen.

10.2 Quantitative Beurteilung der Automatisierung

Nachdem wir im vorangegangenen Abschnitt einige grundlegende Maße eingeführt und die Probleme bei der Messung von Produkteigenschaften diskutiert haben, wollen wir in diesem Teil der Arbeit eine Beurteilung der Automatisierung auf der Basis quantitativer Ergebnisse vornehmen.

Dazu werden die Resultate zweier Fallstudien vorgestellt, miteinander verglichen und die Verbesserungen untersucht und diskutiert. Anschließend wird dem Auto-

matisierungsgewinn der Aufwand für die Erstellung der Automatisierungswerkzeuge gegenübergestellt.

Bevor wir mit der Untersuchung beginnen können, ist es jedoch angebracht, relevante statistische Techniken kurz zu erläutern und deren Anwendbarkeit zu diskutieren.

10.2.1 Statistische Grundlagen

Wie wir bereits bei der Betrachtung des Prozessmaßes zur Bestimmung der Fehleranzahl sahen, stehen für die Beurteilung von Fallstudien über Software-Entwicklungsmethoden nur eine sehr geringe Zahl von Messwerten zur Verfügung. Eine größere Datenbasis ließe sich durch formale Experimente [LaM96, S.7] erhalten, deren Durchführung im Bereich der Software-Entwicklung allerdings sehr kostspielig ist und aus verständlichen Grunde im Rahmen dieser Arbeit nicht durchgeführt werden konnte.

Um aus wenigen Messdaten dennoch relevante Aussagen ableiten zu können (und die Aussagen auch mit einer gewissen Glaubwürdigkeit versehen zu können), bedient man sich der *induktiven Statistik* (zur Einführung siehe z.B. [BaB01, S.133ff.]). Die Techniken der induktiven Statistik erlauben dabei aus den zur Verfügung stehenden Ausprägungen einen charakteristischen aber unbekannten Wert für die Gesamtsituation zu schätzen und Hypothesen zu prüfen. In den folgenden Abschnitten werden wir daher auf die von uns verwendeten Verfahren der induktiven Statistik und die zugehörigen Grundlagen eingehen.

Skalen

Den Merkmalen können unterschiedliche *Skalen* zu Grunde liegen. Wichtig ist die Kenntnis der jeweiligen Skala, da viele statistische Methoden nur dann angewendet werden dürfen, wenn eine bestimmte Art der Skala vorliegt.

In Tabelle 10-2 sind die wichtigsten Skalentypen aufgeführt (vgl. [BaB01, S.6f.] und [Fen91, S.36]):

Tabelle 10-2. Skalen für Merkmale

Art der Skala		**(zusätzl.) Eigenschaften**	**Beispiel**
Nominal-		$x = y$	Sozialversicherungsnummer
Ordinal-		$x > y$	Intelligenzquotient
Kardinal-	Intervall-	$x_1 - x_2 > y_1 - y_2$	Temperatur in Celsius
	Verhältnis-	natürlicher Nullpunkt	Länge
	Absolut-	natürliche Einheit	Stückzahl

Alle bisher vorgestellten Merkmale basieren auf einer Kardinalskala (genauer einer Verhältnisskala, da sich für alle Größen ein Nullpunkt ausmachen lässt). Ordinalskalierte Größen sind typischerweise solche, welche durch subjektive Maße, also durch eine Einschätzung der Entwickler, gemessen werden. Diese werden – wie bereits erläutert – nicht weiter besprochen.

Hypothesentests und Signifikanz

Die Ziele einer Untersuchung werden typischerweise in der Form von Hypothesen formuliert. Dabei wird zwischen *Nullhypothese* H_0 und *Alternativhypothese* H_1 unterschieden (vgl. [BaB01, S.179ff.]). Die Nullhypothese gilt solange als wahr bis Messungen deren Falschheit nachweisen (vgl. [HeH02, S.104]). Die Alternativhypothese ist diejenige Situation, die als wahr erwartet wird, wenn H_0 unwahrscheinlich ist.

Für den von uns betrachteten Vergleich ist die Nullhypothese: „Die Automatisierung bringt keine Produktivitätssteigerung“. Damit wäre eine Alternativhypothese: „Die Automatisierung führt zu einer Produktivitätssteigerung“. Wird die Hypothese H_0 abgelehnt, wird dadurch also H_1 gestützt. Allerdings reicht dies nicht unbedingt aus, um H_1 als wirklich wahr anzuerkennen, denn H_1 ist nur eine von vielen anderen alternativen Hypothesen, die auch zu den beobachteten Ergebnissen hätten beitragen können (siehe [Fen91, S.70]). So könnte eine alternative Hypothese zum obigen Beispiel auch lauten: „Ein besseres Entwicklerteam bringt eine Produktivitätssteigerung“. Es ist aus diesem Grund wichtig, möglichst alle Einflussfaktoren eines Experiments bzw. einer Fallstudie zu kennen und den Einfluss dieser Faktoren auf das Ergebnis zu untersuchen, um die (interne) Validität der Ergebnisse zu garantieren [LaM96, S.9]. Dies fällt zugegebenermaßen nicht immer leicht und ist bei der notwendigen Beteiligung von Menschen in einem Software-Engineering-Experiment nicht immer vollständig möglich.

Ein wichtiges Kriterium, das bei einem Vergleich von Software-Experimenten oder -Fallstudien leider zu oft außer Acht gelassen wird, ist die statistische *Signifikanz* der Resultate. Insbesondere, da die Anzahl der Merkmalsausprägungen häufig sehr gering ist, ist es wichtig zu wissen, wie aussagekräftig eine Beobachtung ist. Um dies beurteilen zu können, muss die Fehlentscheidung einer Ablehnung von H_0 berücksichtigt werden [HeH02, S.97].

Dazu wird eine zulässige Irrtumswahrscheinlichkeit α, auch *Signifikanzniveau* genannt, angegeben. Die typischen Werte für α liegen bei 0,05 oder 0,01 [Fen91, S.70]. Die eigentliche Entscheidung über die Ablehnung der Hypothese erfolgt dann auf der Basis der Wahrscheinlichkeit, dass die gemessenen Merkmalsausprägungen auch entstanden sein könnten, wenn die Nullhypothese wahr wäre. Diese Wahrscheinlichkeit nennt man den p-Wert [HeH02, S.108]. Nur bei einem p-Wert kleiner als α wird H_0 verworfen.

Konfidenz

Für einen ähnlichen Zweck wie die Signifikanz wird die Größe der *Konfidenz* eingesetzt. Hierbei wird allerdings beurteilt, wie groß die Irrtumswahrscheinlichkeit α der Schätzung eines charakteristischen Werts ϑ der Gesamtsituation ist.

Genauer gesagt wird nicht ein Schätzwert $\hat{\vartheta}$, sondern für ein *gegebenes Konfidenzniveau* $1-\alpha$ (*Vertrauenswahrscheinlichkeit*) ein sog. *Konfidenzintervall* in der Form $[\hat{\vartheta}_l, \hat{\vartheta}_r]$ angegeben. Mit einer Wahrscheinlichkeit von $1-\alpha$ liegt dann der charakteristische Wert der Gesamtsituation, also ϑ, in diesem Intervall.

Zufallsauswahl, Normalverteilung und exakter Test

Viele Verfahren der induktiven Statistik basieren auf der Annahme, dass die Messdaten durch eine *zufällige Auswahl* (siehe [BaB01, S.135]) aus einer *normalverteilten* Grundgesamtheit gezogen wurden. Dies ist eine vernünftige Annahmen für z.B. Umfragen zum Wahlausgang. Im Bereich der Software-Experimente ist diese Annahme allerdings selten zu halten [LaM96, S.38]. Zum einen ist so gut wie keine Zufallsauswahl gegeben, zum anderen liegt fast nie eine Normalverteilung zu Grunde auch wenn viele der Merkmale verhältnisskaliert sind (vgl. dazu [Fen91, S.98]). Insbesondere sind die Daten solcher Messungen häufig in Richtung des Nullpunkts verzerrt und beinhalten teilweise sehr große Werte (*Ausreißer*). Bereits in Abb. 10-4 auf Seite 309 lässt sich das Phänomen der Verzerrung beobachten, da sich die meisten Messpunkte bei $C<150$ und $F\leq 1$ befinden.

Man muss bei dem Einsatz statistischer Techniken, die eine Normalverteilung annehmen, daher äußerst vorsichtig sein. Es existieren allerdings auch Techniken, die ohne eine Annahme bzgl. der Verteilung der Grundgesamtheit auskommen (dieses sind die sog. *nicht-parametrischen Verfahren* [HeH02, S.100]). Dazu zählen die im Folgenden vorgestellten *robusten Methoden*.

Die geringe Anzahl von Merkmalsausprägungen hatten wir bereits als Problempunkt ausgemacht. Diese geringe Zahl n der Ausprägungen eines Merkmals führt zusätzlich zu Problemen bei der Bestimmung des p-Werts. Da diese Bestimmung auf der Basis einer Zufallsverteilung erfolgt, die erst ab Werten von n größer 10 (besser größer 30, vgl. [HeH02, S.103]) durch bekannte Verteilungen (wie die Normalverteilung) approximiert werden kann, muss p für kleinere Werte von n aus entsprechenden Tabellen entnommen werden. In einem solchen Fall spricht von einem *exakten Test*.

Korrelation

Zur Quantifizierung des Zusammenhangs zweier Merkmale x und y berechnet man die *Korrelation* $r(x, y)$. Neben der Richtung der Korrelation, welche durch das Vorzeichen von r bestimmt ist, wird auch eine Stärke des Zusammenhangs beschrieben.

Im Allgemeinen liegt r zwischen -1 und 1, wobei bei $r = 1$ ein vollständig positiver Zusammenhang zwischen den Ausprägungen von x und y und bei $r = -1$ ein vollständig negativer Zusammenhang besteht. Bei $r = 0$ sind die Merkmale *unkorreliert*.

Eine bekannter Korrelationskoeffizient, der auch häufig in Publikationen von Software-Experimenten verwendet wird, ist der *Bravais-Pearson*-Korrelationskoeffizient. Hierzu wird die normierte Summe der Produkte der Abweichungen der Merkmalsausprägungen x und y von deren jeweiligem Mittelwert $\bar{x}$ bzw. $\bar{y}$ berechnet (vgl. auch [BaB01, S.36f.]):

$$r = \frac{\sum (x_i - \bar{x}) \cdot (y_i - \bar{y})}{\sqrt{\sum (x_i - \bar{x})^2 \cdot \sum (y_i - \bar{y})^2}}$$

Eine Voraussetzung für die Anwendung dieses Koeffizienten ist eine Kardinalskalierung der Merkmale. Wie wir bereits erläuterten, ist dies i.d.R. gegeben. Allerdings darf Bravais-Pearson nur bei normalverteilten Merkmalen eingesetzt werden, was wie oben dargelegt wurde, in den meisten Software-Experimenten nicht der Fall ist.

Robustere Maße des Zusammenhangs sind der *Rangkorrelationskoeffizient* von *Spearman* (ρ) und der Korrelationskoeffizient von *Kendall* (τ). Diese Koeffizienten sind sowohl gegenüber atypischen Werten (Ausreißern) als auch nicht-linearen Zusammenhängen robust (vgl. [Fen91, S.1-2f.]).

In dieser Arbeit werden wir ρ als Koeffizienten wählen, welchen man sich als Bravais-Pearson-Korrelationskoeffizienten vorstellen kann, welcher auf der Basis von Rängen an Stelle von konkreten Merkmalswerten berechnet wurde (siehe dazu auch [BaB01, S.38f.]). Dazu werden die Merkmalswerte in eine Rangordnung gebracht und r mit Hilfe deren Rangnummern berechnet. Existieren mehrere gleiche Merkmalswerte, so wird deren Rang als arithmetisches Mittel der jeweiligen Rangnummern festgelegt (siehe [HeH02, S.121]).

Um die Vertrauenswürdigkeit eines solchen Korrelationskoeffizienten beurteilen zu können, setzen wir das oben beschriebene Konzept der Signifikanz ein. Dabei definieren wir als Nullhypothese H_0: $\rho = 0$, d.h. wir nehmen an, dass die Merkmale nicht korreliert sind, und entsprechend als Alternativhypothese H_1: $\rho \neq 0$, d.h. wir nehmen eine Korrelation zwischen x und y an. Bei den Ergebnissen der folgenden Fallstudien werden wir die konkreten Ausprägungen von ρ dann jeweils mit „**“ versehen, falls bei $\alpha = 0{,}01$ gilt: p-Wert $< \alpha$-Niveau (siehe oben). Dies entspricht der üblicherweise von Statistik-Software-Paketen verwendeten Notation. Für unsere Auswertungen setzten wir neben selbst implementierten Werkzeugen auch die kommerziellen Pakete SPSS 6.1 und StatXact 6 ein. Letzteres Werkzeug erlaubt die exakte Berechnung nicht-parametrischer Größen.

Trotz der Robustheit dieser Methoden sollte ein erster Schritt in der Beurteilung von Messergebnissen stets deren grafische Interpretation (z.B. an Hand eines Streuungsdiagramms wie in Abb. 10-4 auf Seite 309) sein, da grobe Ausreißer auch die robusten Techniken beeinflussen können (vgl. [Fen91, S.103]).

Regression

Wurde eine Korrelation zwischen zwei Merkmalen identifiziert, so kann man daran gehen, die exakte Natur dieses Zusammenhangs durch das Verfahren der *Regression* zu bestimen. Dazu kann unter anderem die Signifikanz des linearen Zusammenhangs der beiden Größen x und y geprüft werden [HeH02, S.266]. Eine der beiden Größen wird dabei als die abhängige Variable und die andere Größe als die unabhängige Variable gewählt [BaB01, S.42]. Nehmen wir x als unabhängige Variable an, dann wird ein linearer Zusammenhang beschrieben durch

$$\hat{y} = b_0 + b_1 x \qquad \textbf{(Gleichung 10-4)}$$

Die Werte von b_0 und b_1 werden üblicherweise mit der Methode der kleinsten Quadrate bestimmt, d.h. man berechnet b_0 und b_1 so, dass die Summe der Quadrate der Abweichungen d_i mit $d_i = y_i - \hat{y}(x_i)$ minimiert wird [BaB01, S.42f.].

Die Methode der kleinsten Quadrate setzt keine Normalverteilung von x und y voraus. Allerdings muss für die Signifikanzanalyse die Normalverteilung von d vorausgesetzt werden. Diese Annahme kann leider in vielen Fällen nicht eingehalten werden. Wie schon für die Korrelation führen wir daher eine robuste Methode zur Berechnung der linearen Regression ein, die Kendall-Theil-Regression (siehe [Fen91, S.104] und [HeH02, S.266ff.]).

Theil schlägt zur Berechnung von b_1 vor, den *Median* [BaB01, S.17] der Steigungen aller Kombinationen von Punkten (x_i, y_i) und (x_j, y_j) mit $x_i \neq x_j$ zu berechnen. Die Steigungen $b_{1,ij}$ sind dabei:

$$b_{1,ij} = \frac{y_j - y_i}{x_j - x_i}$$

Die Bestimmung von b_0 erfolgt analog durch die Berechnung des Medians aller Werte $b_{0,i}$ mit

$$b_{0,i} = y_i - b_1 x_i .$$

Eine alternative Bestimmung von b_0 wird in [HeH02, S.267] vorgeschlagen und basiert auf der Verwendung der Mediane von x und y:

$$b_0 = \mathrm{Med}(y) - b_1 \cdot \mathrm{Med}(x) .$$

Wir werden im weiteren Verlauf die letztere Berechnungsvorschrift verwenden, da sie analog zu der Methode der kleinsten Quadrate die Eigenschaft besitzt, dass die Regressionsgerade durch den Punkt $(\mathrm{Med}(x), \mathrm{Med}(y))$ verläuft. Desweiteren ist sie robuster als die vorhergehende Lösung (vgl. auch [HeH02, S.267]).

Eine schöne Eigenschaft dieser Regressionsberechnung ist, dass die Signifikanz des Hypothesentests für H_0: $b_1 = 0$ (also die Annahme keiner linearen Beziehung) auf die Signifikanz des Kendall-Korrelationskoeffizienten τ (also H_0: $\tau = 0$) zurückgeführt werden kann [HeH02, S.266]. Daher kennzeichnen wir einen signifikanten linearen Zusammenhang zwischen x und y wieder mit „**“.

Vergleich unabhängiger Daten

Um den Automatisierungsgewinn zu beurteilen, müssen wir in den weiter unten vorgestellten Fallstudien einen Vergleich zweier unabhängiger Gruppen von Daten durchführen. Auch hierbei ist es wieder angebracht, eine grafische Darstellung der Ausprägungen zu erstellen, um zunächst ein Gefühl für den Unterschied der Daten zu erhalten. Eine sinnvolle Darstellung bieten *Box-Plots* [HeH02, S.25][Fen91, S.99], welche neben dem Median der Daten auch deren Ausdehnung (25%- und 75%-Quantile [BaB01, S.119]), deren Verschiebung und eventuelle Ausreißer (durch „*“ und „o“ gekennzeichnet) darstellen. In Abb. 10-9 ist ein solcher Plot gezeigt.

Nehmen wir nun an, dass die erste Gruppe u kleinere Werte der Merkmalsausprägungen als die zweite Gruppe v aufweist, z.B. eine geringere Produktivität, dann formulieren wir nach [HeH02, S.118] die Nullhypothese

$$H_0\colon \mathrm{Prob}(u > v) = 0{,}5\,,$$

d.h. in genau 50% der Fälle ist u größer als v und demzufolge hätte v ebenso aus der Gruppe u stammen können. Unsere Alternativhypothese lautet

$$H_1\colon \mathrm{Prob}(u > v) < 0{,}5\,,$$

d.h., dass die Wahrscheinlichkeit größer als $1/2$ ist, dass u kleiner als v ist.

Als nicht-parametrische Methode zur Widerlegung von H_0 bietet sich der *Wilcoxon-Mann-Whitney-Rangsummentest* an (siehe z.B. [HeH02, S.118]). Dazu nehmen wir o.b.d.A. an, dass n die Anzahl von Merkmalsausprägungen von u sei und m die Anzahl der Merkmalsausprägungen von v mit $n \le m$.

Zunächst berechnet man für jedes u_i und u_j den gemeinsamen Rang R_k, d.h. den Rang innerhalb der Menge $\{u_1, \ldots, u_n, v_1, \ldots, v_m\}$. Die Rangsumme W ergibt sich dann zu

$$W = \sum_{k=1}^{n} R_k\,. \qquad \textbf{(Gleichung 10-5)}$$

Die Nullhypothese kann bei einem Signifikanzniveau von α abgelehnt werden, wenn gilt

$$W \leq x^*(\alpha, n, m),$$

wobei die konkreten Werte der Verteilung x^* des Wilcoxon-Mann-Whitney-Tests für einen exakten Test (bei $n, m \leq 10$) einer Tabelle zu entnehmen sind [HeH02, S.461f.] und für größere Mengen von Merkmalsausprägungen auf Basis der Normalverteilung abgeschätzt werden können (vgl. [HeH02, S.121ff.]).

Zuletzt kann – analog zur Korrelation und der anschließenden Regression – auch für den Vergleich zweier Datensätze eine Quantifizierung des Unterschieds erfolgen. Eine Technik ist hierbei die Berechnung des *Hodges-Lehman-Schätzers* $\hat{\Delta}$, welcher robuster ist als die reine Differenz der Mediane von u und v. Der Schätzer ist definiert als der Median aller paarweisen Differenzen zwischen Werten aus u und v, also:

$$\hat{\Delta} = \text{Med}(u_i - v_j) \text{ für } i = 1, \ldots, n \text{ und } j = 1, \ldots, m.$$

Um die Variationsbreite dieser Schätzung beurteilen zu können, lässt sich zusätzlich ein Konfidenzintervall (siehe oben) bestimmen (vgl. [HeH02, S.133]).

10.2.2 Experimentelle Untersuchung des Automatisierungsgewinns

Zur Beurteilung des Produktivitätsgewinns, der sich durch unseren Automatisierungsansatz erreichen lässt, wollen wir die Ergebnisse zweier Fallstudien vergleichen. In der ersten Fallstudie *FloorAutomationX* wurde von der Automatisierung kein Gebrauch gemacht, wohingegen in der Fallstudie *RoomAutomationX* die Werkzeuge Parser, Unparser und Checker zum Einsatz kamen.

Der Vergleich wird dabei mit Hilfe der oben vorgestellten statistischen Methoden erfolgen. Bevor wir die Resultate vorstellen, soll hier zunächst die allgemeine Vorgehensweise dargelegt werden.

Die HTML- und die SDL-Dokumente der Control-Object-Types wurden getrennt betrachtet, da die Größe der Artefakte bei den SDL-Dokumenten anders in den Aufwand eingeht als bei den HTML-Dokumenten (siehe Regressionsgeraden in Abb. 10-5 auf Seite 319 und Abb. 10-6 auf Seite 322). Dies ist nicht verwunderlich, wurde oben doch der Einfluss der „Programmiersprache“ auf die Ausprägung des Größenmaßes bereits erwähnt. Würde man nun beide Dokumenttypen gleichzeitig betrachten, so würde diese Tatsache ignoriert. Wir werden in dieser Arbeit daher zunächst kategorisierte Ergebnisse vergleichen. Dies entspricht dem statistischen Konzept der Paarbildung (engl. „Blocking“), um den Fehler zu reduzieren (siehe [BHH78, S.105] und [Fen91, S.66]).

Da sich die *Produktivität* p (auch als *Effizienz* bezeichnet), als Quotient von Output zu Input [Gab98][Fen91, S. 261], entweder mit Hilfe der Größe S zu $p = S/E$ oder mit Hilfe der Komplexität C zu $p = C/E$ berechnen lässt, muss zunächst eine Auswahl aus diesen beiden Merkmalen erfolgen. Die Literatur beschreibt Experimente, bei welchem der Schluss gezogen werden kann, dass die Merkmale Komplexität und Größe prinzipiell austauschbar verwendet werden können (siehe z. B. [She93, S. 208ff.]). Auch in den von uns betrachteten Fallstudien besteht zwischen der Größe und der Komplexität der SDL-Artefakte eine sehr gute Korrelation von $\rho = 0{,}94^{**}$ (siehe Tabelle 10-4 auf Seite 321) bzw. $\rho = 0{,}93^{**}$ (siehe Tabelle 10-8 auf Seite 325). In der Fallstudie *RoomAutomationX* stellten wir zusätzlich eine Korrelation von $\rho = 0{,}93^{**}$ für die HTML-Dokumente fest.

Da S einfacher zu berechnen ist und die Komplexität C für die HTML-Dokumente von *FloorAutomationX* nicht automatisch ermittelt werden konnte (die konkrete Syntax weicht von der in dieser Arbeit definierten Syntax ab), wählen wir im folgenden die Größe S als Basis der Produktivitätsberechnung.

Konsolidierung der Messdaten

Um die Variationen der Produktivität innerhalb der Artefakte berücksichten zu können und keine Verfälschungen durch die Reduktion auf eine einzige Produktivitätsgröße zu erhalten, werden wir die Produktivität p für jedes Artefakt $a \in A$ einzeln berechnen.

Hierbei kommt es allerdings zu Problemen. Das erste Problem ist, dass Ausreißer ($A_{\text{ausr}} \subseteq A$) in den Messdaten zu erkennen sind (siehe bei den einzelnen Fallstudien), bei welchen der Aufwand eines Artefakts nicht auf dessen Größe zurückgeführt werden kann. Diese werden daher aus der Betrachtung herausgenommen.

Das zweite Problem ist, dass von den Dokumenttypen wie z. B. der Taskliste nur eine Instanz existiert (A_{einzeln}) und daher ein relevanter Vergleich statistisch fraglich ist. Als Lösung werden wir die Aufwände der einzelnen Dokumentinstanzen A_{einzeln} anteilsmäßig auf alle Control-Object-Type-Dokumente verteilen.

Wir gehen also nach dem folgenden Schema vor:

1. Feststellen einer signifikanten Korrelation zwischen S und E.
2. Berechnung der Regressionsgerade $\hat{E} = b_0 + b_1 \cdot S$ und der Signifikanz der Steigung b_1 für alle Artefakte $a \notin A_{\text{ausr}} \cup A_{\text{einzeln}}$.
3. Bestimmung des Aufwands E_{rest}, der sich aus den nicht berücksichtigten einzelnen Dokumentinstanzen ergibt:

$$E_{\text{rest}} = \sum_{a \in A_{\text{einzeln}}} E(a)$$

4. E_{rest} anteilsmäßig (in Abhängigkeit des Zusammenhangs der erfassten Aufwandsdaten) auf alle Control-Object-Types (auch die Ausreißer) verteilen. Dazu berechnet man zunächst die Summe aller hypothetischen Aufwände $\hat{E}_{\text{sum}} = \sum \hat{E}(a)$. Die „konsolidierten" Aufwände E' ergeben sich dann zu:

$$E'(a) = E(a) + \frac{\hat{E}(a)}{\hat{E}_{\text{sum}}} \cdot E_{\text{rest}}$$

Um letztlich einen gerechten Vergleich der Ergebnisse unterschiedlicher Fallstudien vornehmen zu können, müssen neben der Größe (oder Komplexität) der entwickelten Artefakte und dem Zeitaufwand dieser Entwicklung auch die „Qualität" der Produkte (insbesondere deren Korrektheit und Zuverlässigkeit) berücksichtigt werden. Die Produktqualität ist insofern wichtig, als dass man eigentlich nur bei Produkten eines identischen Qualitätsniveaus einen Vergleich der Produktivität vornehmen darf. Ohne die Berücksichtigung dieser dritten Messgröße, würde man ansonsten eine maximale Verbesserung erzielen, wenn das Vergleichsprodukt in möglichst Null Zeit entwickelt wurde.

Der korrekte Vergleich zweier Produkte (wenn man neben den obigen Größen auch die anderen Qualitätsmerkmale aus Abschnitt 7.1.1 auf Seite 177 heranzieht) ist ein allgemeines Problem, wie auch Witt in [Wit87] herausstellt. Wir werden aber versuchen, auf der Basis der von uns messbaren Größen einen halbwegs fairen Vergleich zu realisieren. Hierzu verwenden wir den in Abschnitt 10.1.1 auf Seite 302 definierten Konsistenzquotienten κ, mit welchem wir die *Qualitäts-Produktivität* q definieren als

$$q = \frac{\kappa \cdot S}{E} = \kappa \cdot p\,.$$

Damit erreicht nur ein Artefakt, das keine Inkonsistenzen aufweist, die höchstmögliche Produktivität bei seiner Erstellung. In allen anderen Fällen nimmt die Qualitätsproduktivität linear mit der Anzahl der Inkonsistenzen ab, bis q für ein völlig sinnloses Dokument bei 0 angekommen ist.

Wie auch schon für die Berechnung der Komplexität, können wir κ leider nicht für die SDL-Dokumente der Fallstudien *FloorAutomationX* berechnen. Dies stellt in unsere Augen aber kein allzu kritisches Problem dar, da die SDL-Dokumente auch bereits in dieser Fallstudie mit Hilfe des Syntax- und „Well-formedness"-Analysators der Telelogic Tau SDL Suite überprüft wurden und fehlerfrei sind. Wir können daher ein $\kappa = 1$ annehmen.

Fallstudie *FloorAutomationX*

Die Fallstudie *FloorAutomationX*, die von 8 Entwicklern [MeQ01] durchgeführt wurde, hatten wir bereits in Abschnitt 8.2.2 auf Seite 224 eingeführt (vgl. auch [QuZ99] und [Que02, S.341]). Wir wollen hier daher lediglich deren quantitativen Daten vorstellen, da diese als Referenz („Base-Line“) für den Vergleich mit der von uns durchgeführten Fallstudie *RoomAutomationX* dient (siehe nachfolgender Abschnitt).

Wir beginnen mit den Resultaten bzgl. der HTML-Objekttypen. In Abb. 10-5 ist das Streuungsdiagramm dieser Dokumente gezeigt.

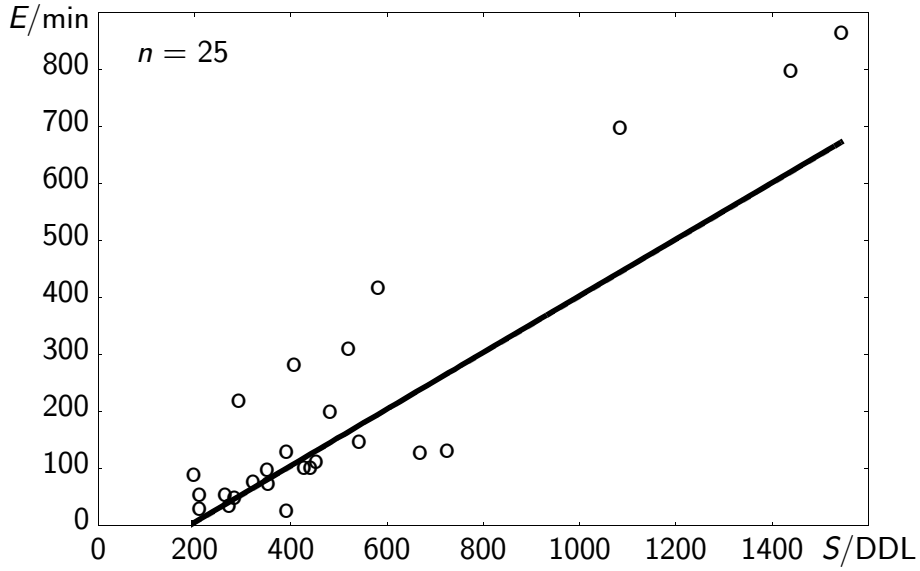


Abbildung 10-5. Streuungsdiagramm für HTML-Dokumente von *FloorAutomationX*

Es gab hierbei keine nennenswerten Ausreißer, weshalb wir den Korrelationskoeffizienten ρ auf der gezeigten Menge von Daten berechneten. Es ergibt sich eine Spearman-Korrelation von $\rho = 0{,}79^{**}$. Damit ist eine signifikante Korrelation nachgewiesen und wir können die Bestimmung der Regressionsgerade vornehmen (siehe Schritt 2 von oben).

Wir erhalten für die Steigung $b_1 = 0{,}496^{**}$ und damit einen signifikanten linearen Zusammenhang der Form $\hat{E} = 93{,}5 + 0{,}496 \cdot S$. In Abb. 10-5 ist der Verlauf der Regressionsgeraden gezeigt.

Der Aufwand, den es nun auf die HTML-Objekttypen (nach Schritt 3 von oben) zu verteilen gilt, setzt sich aus dem Aufwand der Spezifikation der Objektstruktur, der Taskliste und einigen Testfällen zusammen. Insgesamt ist $E_{\text{rest}} = 3160\,\text{min}$, die Summe der hypothetischen Aufwände ist $\hat{E}_{\text{sum}} = 4032\,\text{min}$.

Die konsolidierten Aufwandsdaten sind in Tabelle 10-3 gezeigt.

Tabelle 10-3. Daten für HTML-Objekttypen von *FloorAutomationX*

Objekttyp	**Konsolidierter Aufwand** E'/min	**Konsistenz-quotient** κ	**Produktivität** $p/\mathrm{DDL}\cdot\mathrm{min}^{-1}$	**Qualitäts-Produktivität** $q/\mathrm{DDL}\cdot\mathrm{min}^{-1}$
Blind	81,43	0,84	3,46	2,91
Contact	100,45	0,13	3,88	0,50
Desk	124,60	0,59	2,58	1,52
Dimmer	79,04	0,69	3,33	2,30
Door	253,93	0,68	1,15	0,78
Floor	335,39	0,94	2,16	2,03
Hallway	565,76	0,14	1,03	0,14
HWDoor	309,85	0,31	1,55	0,48
HWLight	363,67	0,55	1,12	0,61
HWOcc	434,63	0,06	1,19	0,07
Light	280,19	0,65	1,93	1,26
MotDet	204,45	0,53	1,91	1,01
OutDoorLight	190,84	0,69	2,24	1,54
PIDCtrl	62,15	0,83	4,36	3,62
PulseGen	34,42	0,50	6,10	3,05
Radiator	210,57	0,80	2,15	1,72
Room	1042,44	0,89	1,04	0,93
RoomLt	1388,02	0,96	1,11	1,07
RoomOcc	156,89	0,84	2,23	1,87
RoomTm	1280,17	0,97	1,12	1,09
Section	310,60	0,86	2,15	1,85
SunDet	133,66	0,36	2,63	0,95
TaskLight	88,75	0,67	2,23	1,49
TempSens	195,90	0,82	2,25	1,84
Valve	59,42	0,93	3,53	3,29
	$\sum = 8287$	Med = 0,69	Med = 2,16	Med = 1,49

Neben den Aufwandsdaten sind auch die Ausprägungen des Konsistenzquotienten κ und die Produktivität ohne und mit Einbeziehung von κ gezeigt. Hierbei fällt auf, dass κ oft relativ kleine Werte (z.B. 0,14 bei Hallway) annimmt. Dies liegt darin begründet, dass die Modellierer von Fallstudie *FloorAutomationX* die Verfolgbar-

keitsrelationen leider nicht konsequent angaben (so z. B. die Verfolgbarkeit zwischen Needs und Tasks). Erst in einem späteren Schritt, dessen zeitlicher Aufwand nicht erfasst wurde, wurde dieser Makel korrigiert. Allerdings konnte dabei nicht immer eindeutig festgestellt werden, welche Anforderungen die ursprünglichen Modellierer jeweils realisierten.

Nach der Betrachtung der HTML-Artefakte wollen wir uns nun den SDL-Dokumenten widmen. Dazu sind in Tabelle 10-4 zunächst die Korrelationskoeffizienten für den Aufwand, die Komplexität und die Größe gezeigt.

Tabelle 10-4. Korrelation der SDL-Objekttypen von *FloorAutomationX*

Spearman-Korrelation ρ	**Komplexität** C	**Aufwand** E
Aufwand E	0,91**	
Größe S	0,94**	0,94**

Die Ausreißer Floor, Section und dessen Varianten Section1 bis Section3 wurden nicht berücksichtigt, da diese einen zu geringen Aufwand im Vergleich zu deren Größe auswiesen. Dies liegt darin begründet, dass ab einem gewissen Entwicklungsfortschritt die Aufwandserfassung nicht mehr für die jeweils bearbeiteten Control-Object-Types erfolgte, sondern für das Gesamtsystem (RequirementsSpecification).

Neben der Elimination dieser Ausreißer wurde zusätzlich der Gewinn, der bei der Variantenbildung durch die Wiederverwendung [Que02, S. 219ff.] eines Ausgangsobjekttyps erreicht wurde, kompensiert. Dies bedeutet, dass die Ergebnisse die Aufwände widerspiegeln, die ohne eine solche Wiederverwendung nötig gewesen wären und daher nicht direkt mit den von Queins in [Que02] genannten Zahlen verglichen werden können.

Abb. 10-6 zeigt das Streuungsdiagramm der SDL-Objekttypen, wobei die Ausreißer durch ein „x“ dargestellt sind. Auch hier bestimmen wir zunächst wie bei den HTML-Artefakten die Regressionsgerade (ohne Berücksichtigung der Ausreißer). Wir erhalten eine signifikante Steigung von $b_1 = 1{,}449^{**}$ und berechnen damit eine Regressionsgerade von $\hat{E} = -376 + 1{,}449 \cdot S$.

Wegen des y-Achsenabschnitts von $b_0 < 0$ kann sich ein negativer hypothetischer Aufwand (für $S < 260\,\mathrm{DDL}$) ergeben, was nicht sinnvoll ist. Für solche Messdaten verwenden wir daher einen hypothetischen Aufwand, der dem Median der Aufwände aller Datenpunkte entspricht, die ein $\hat{E} < 0$ hätten. Es ergibt sich hierbei ein Wert von 63 Minuten.

Damit können wir nun mit der Konsolidierung der Daten weiterfahren. Als Aufwand E_{rest} ergeben sich 2841 Minuten, was den Aufwand für das Testen und die Inspektion des Gesamtsystems und die Spezifikation einiger SDL-Hilfsfunktionen beinhaltet. Die Summe der hypothetischen Aufwände ist $\hat{E}_{\mathrm{sum}} = 35542\,\mathrm{min}$.

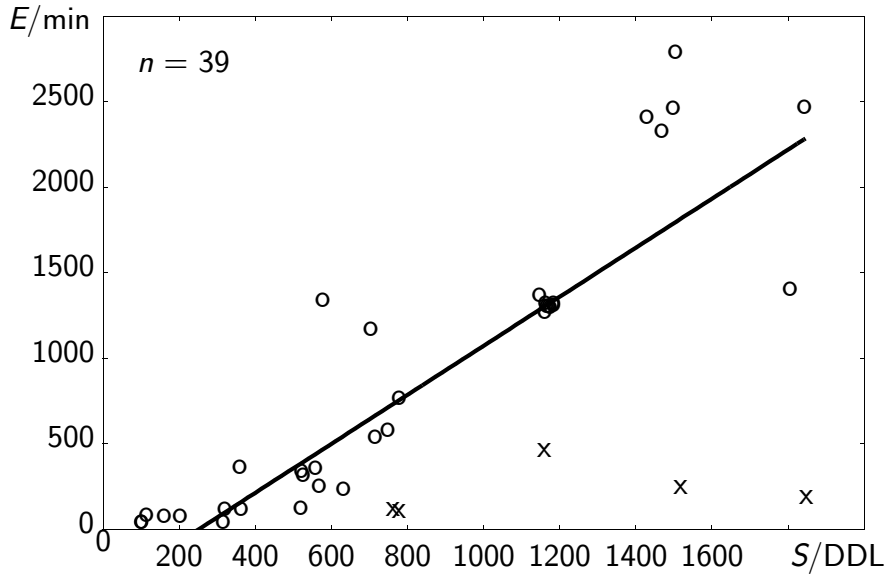


Abbildung 10-6. Streuungsdiagramm für SDL-Dokumente von *FloorAutomationX*

Tabelle 10-5 zeigt die konsolidierten Aufwände E', wobei p mit q übereinstimt, da wir von einer vollständigen Konsistenz der SDL-Dokumente ausgehen können.

Tabelle 10-5. Daten für SDL-Objekttypen von *FloorAutomationX*

Objekttyp	**Aufwand** E'/min	**Prod.** $q/\text{DDL} \cdot \text{min}^{-1}$	**Objekttyp**	**Aufwand** E'/min	**Prod.** $q/\text{DDL} \cdot \text{min}^{-1}$
Blind	1362,66	0,42	OutDoorLight	113,87	3,18
Cl1D3W	1406,96	0,84	PIDCtrl	68,04	2,97
Cl2D2W	1395,57	0,84	PulseGen	35,04	2,85
Cl2D9W	1572,01	1,15	Radiator	358,29	1,45
Cl3D2W	1412,53	0,82	Room	1360,30	0,85
Contact	108,89	2,93	RoomLt	2921,03	0,51
Desk	378,46	1,47	RoomOcc	333,75	1,57
Dimmer	77,04	1,47	RoomTm	2455,97	0,60
Door	139,06	3,73	RoomTm2R	2532,34	0,56
Floor (x)	562,07	2,06	RoomTm3R	2595,33	0,58
Hallway	811,94	0,96	RoomTm9R	2641,29	0,70
HWDoor	623,47	1,20	Section (x)	159,82	4,86
HWLight	263,03	2,40	Section1 (x)	385,53	3,93
HWOcc	576,64	1,24	Section2 (x)	368,87	5,01
Light	273,62	2,07	Section3 (x)	167,97	4,52
MotDet	37,43	8,42	SunDet	1207,25	0,58
Off1D2W	1459,68	0,79	TaskLight	35,04	2,85

Tabelle 10-5. Daten für SDL-Objekttypen von *FloorAutomationX*

Objekttyp	**Aufwand** E'/min	**Prod.** $q/\text{DDL} \cdot \text{min}^{-1}$	**Objekttyp**	**Aufwand** E'/min	**Prod.** $q/\text{DDL} \cdot \text{min}^{-1}$
Off1D3W	1417,96	0,83	TempSens	363,41	0,99
Off2D2W	1392,92	0,84	Valve	70,04	2,28
Off3D2W	1397,88	0,83	ohne (x)	$\sum = 33198$	Med = 1,07

Fallstudie *RoomAutomationX*

In der von Krämer [Krä04] und Walke [Wal04] im Rahmen zweier Projektarbeiten durchgeführten Fallstudie *RoomAutomationX* wurde auf der Basis der Problembeschreibung unseres *RoomAutomation*-Systems aus Abschnitt 6.1.3 auf Seite 138 eine vollständige Systemspezifikation unter Zuhilfenahme der Automatisierungswerkzeuge (insbesondere des Parsers, Unparsers und Checkers) durchgeführt. Wie auch die Entwickler in der Referenzfallstudie besaßen Krämer und Walke nur eine gewisse Einarbeitung in die PROBAnD-Methode, weshalb wir „Erfahrung“ als Quelle einer Produktivitätssteigerung ausschließen können. Auch bezüglich der bereits in *FloorAutomationX* eingesetzten Entwicklungswerkzeuge (HTML-Editoren und SDL-Modellierungswerkzeug) gab es keine relevanten Änderungen, die sich in einer Produktivitätssteigerung hätten niederschlagen können.

Neben der formalen Spezifikation des Steuer- und Regelsystems für Temperatur, Beleuchtung und Blendung wurde auch ein entsprechender Raum-Simulator spezifiziert, der als Testumgebung für das Steuerungssystem diente. Der Steuerungsteil besteht dabei aus 10 Control-Object-Types, welche 13 Tasks realisieren. Der Simulationsteil nutzt zur Implementierung von 13 Tasks 12 Objekttypen. Wenn wir nichts anderes angeben, werden wir bei den folgenden Auswertungen stets die Daten der beiden Teilsysteme zu einem Datensatz zusammenfassen, um eine relevante Anzahl von Messwerten zu erhalten ($n = 10 + 12 = 22$).

Auch in dieser Fallstudie gehen wir nach dem weiter oben beschriebenen Schema vor und betrachten zunächst die Korrelation der relevanten Größen der HTML-Dokumente. Anders als in der Fallstudie *FloorAutomationX* können wir hier die zyklomatische Komplexität der Artefakte bestimmen und erhalten daher die in Tabelle 10-6 gezeigten Koeffizienten.

Tabelle 10-6. Korrelation der HTML-Objekttypen von *RoomAutomationX*

Spearman-Korrelation ρ	**Komplexität** C	**Aufwand** E
Aufwand E	0,74**	
Größe S	0,93**	0,89**

Als einziger Ausreißer wurde der Objekttyp OutIllum (der Simulator-Seite) ausgelassen, da dessen Aufwand zu hoch im Verhältnis zu dessen Größe war. Der Grund war eine von den Entwicklern missverständlich implementierte Ankopplung der Steuerungsseite an die Simulation und deren notwendige Korrektur. In Abb. 10-7 ist das Streuungsdiagramm der Daten gezeigt.

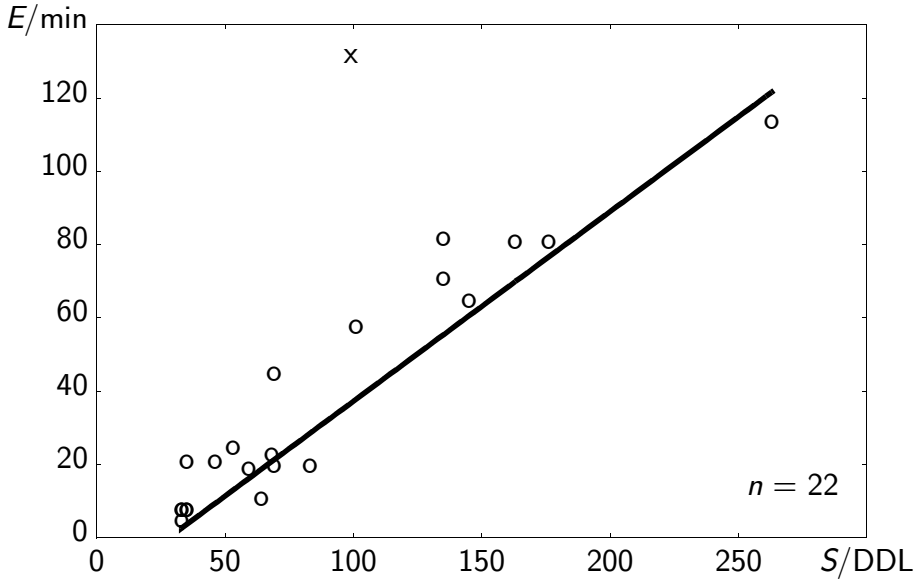


Abbildung 10-7. Streuungsdiagramm für HTML-Dokumente von *RoomAutomationX*

Für die Regression erhalten wir eine signifikante Steigung von $b_1 = 0{,}517^{**}$ mit einem y-Achsenabschnitt von $b_0 = -14{,}5$. Die Gerade ist in Abb. 10-7 dargestellt.

Damit lässt sich nun der verbleibende Aufwand wieder entsprechend aufteilen. Insgesamt ist ein Aufwand von $E_{\text{rest}} = 332$ Minuten zu verteilen, der den Aufwand zur Spezifikation der Taskliste und der Objektstruktur beinhaltet. Insbesondere ist ein Aufwand von 180 Minuten in E_{rest} berücksichtigt, der durch die Anwendung der Automatisierungswerkzeuge während der Spezifikation der HTML-Dokumente entstand. Auch wenn die automatisierten Entwicklungsaktivitäten nun durch das Werkzeug in Null Entwicklerzeit durchgeführt werden, so muss der Aufruf der Werkzeuge dennoch manuell erfolgen und die Ausgaben der Werkzeuge (z. B. Constraint-Verletzungen) überprüft werden. Die Summe der hypothetischen Aufwände war $\hat{E}_{\text{sum}} = 681\,\text{min}$. Damit erhalten wir die in Tabelle 10-7 gezeigten konsolidierten Aufwände.

Da die HTML-Dokumente jeweils mit dem Parser und dem Checker geprüft wurden, erhielten sie keine Inkonsistenzen und deswegen hatten alle Dokumente ein κ von eins. Damit entspricht die Qualitäts-Produktivität q in Tabelle 10-7 exakt der Produktivität p.

Tabelle 10-7. Daten für HTML-Objekttypen von *RoomAutomationX*

Mess- und Steuerteil			Simulator		
Objekttyp	**Aufwand** E'/min	**Prod.** $q/\text{DDL} \cdot \text{min}^{-1}$	**Objekttyp**	**Aufwand** E'/min	**Prod.** $q/\text{DDL} \cdot \text{min}^{-1}$
BlindAct	5,26	6,27	BlindAct	54,35	1,27
IllumSens	21,76	1,61	IllumSens	29,35	2,35
LightAct	8,26	4,00	LightAct	30,31	1,75
LightControl	117,35	1,50	MotionSens	32,88	2,52
MotionSens	8,76	3,99	OutIllum (x)	148,92	0,66
Occupancy-Control	32,09	2,12	OutTemp-Sens	75,42	1,34
RadiatorAct	8,26	4,00	RadiatiorAct	24,54	1,87
Room	19,08	3,35	Room	108,00	1,25
Temperatu-reControl	114,07	1,43	RoomIllum	93,53	1,55
TempSens	8,76	3,99	RoomTemp	172,31	1,53
			TempSens	25,82	2,28
			Wall	97,00	1,39
			ohne (x)	$\sum = 1087$	Med = 1,87

Nun können wir die SDL-Dokumente betrachten, deren Korrelationskoeffizienten in Tabelle 10-8 gezeigt sind.

Tabelle 10-8. Korrelation der SDL-Objekttypen von *RoomAutomationX*

Spearman-Korrelation ρ	**Komplexität** C	**Aufwand** E
Aufwand E	0,78**	
Größe S	0,93**	0,80**

Auch hier müssen wir wieder die Daten des Control-Object-Types OutIllum als Ausreißer herausnehmen. Der Grund war neben der bereits für das HTML-Dokument erwähnten Ursache ein Fehler in der Implementierung des Parser-Werkzeugs, der einen Teil der Entwicklungsinformation auslöschte.

Weiter lassen wir die Daten von Room und OccupancyControl (des Steuerungsteils) weg, da diese einen zu geringen Aufwand im Vergleich zu deren Größe aufweisen. Begründen können wir diese Diskrepanz mit der Tatsache, dass sämtliche Entwicklungsinformationen dieser SDL-Dokumente bereits in den HTML-Dokumenten enthalten waren und damit so gut wie kein manueller Aufwand für deren Erstellung

nötig wurde. Da wir eine vernünftige Korrelation für unser weitergehendes Vorgehen benötigen lassen wir diese dennoch weg. Der errechnete Produktivitätsgewinn wird dadurch höchstens geschmälert.

Wir erhalten nach Elimination dieser Ausreißer das Streuungsdiagramm, das in Abb. 10-8 gezeigt ist.

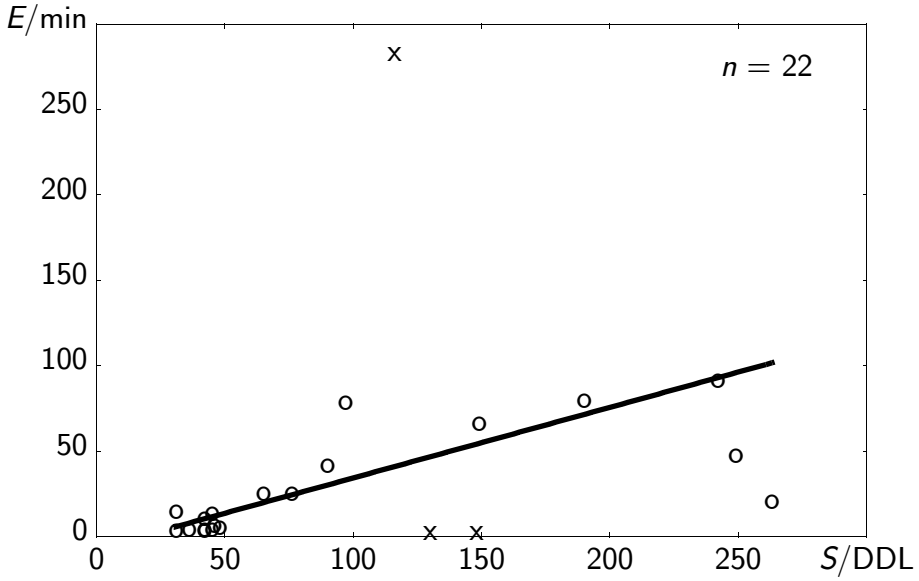


Abbildung 10-8. Streuungsdiagramm für SDL-Dokumente von *RoomAutomationX*

Die Steigung der Regressionsgerade weist mit $b_1 = 0{,}413^{**}$ einen signifikanten Wert auf. Der y-Achsenabschnitt beträgt $b_0 = -6{,}82$. Die Gerade ist in Abb. 10-7 dargestellt.

Insgesamt muss ein Restaufwand von $E_{\text{rest}} = 592$ Minuten anteilsmäßig auf die Objekttypen verteilt werden. Dabei setzt sich E_{rest} aus dem Aufwand zum Test des Gesamtsystems (Prototyptest) und dem Aufwand von 249 Minuten zum Aufruf der Automatisierungswerkzeuge zusammen. Die hypothetischen Aufwände ergaben in der Summe $\hat{E}_{\text{sum}} = 766\,\text{min}$, womit sich die in Tabelle 10-9 gezeigten konsolidierten Aufwandsdaten ergeben.

Tabelle 10-9. Daten für SDL-Objekttypen von *RoomAutomationX*

Mess- und Steuerteil			**Simulator**		
Objekttyp	**Aufwand** E'/min	**Prod.** $q/\text{DDL} \cdot \text{min}^{-1}$	**Objekttyp**	**Aufwand** E'/min	**Prod.** $q/\text{DDL} \cdot \text{min}^{-1}$
BlindAct	17,54	1,77	BlindAct	39,39	1,65
IllumSens	10,05	4,18	IllumSens	13,97	3,44
LightAct	9,14	3,94	LightAct	21,01	2,14
LightControl	161,88	1,49	MotionSens	63,37	1,42

Tabelle 10-9. Daten für SDL-Objekttypen von *RoomAutomationX*

Mess- und Steuerteil			**Simulator**		
Objekttyp	**Aufwand** E'/min	**Prod.** $q/\text{DDL} \cdot \text{min}^{-1}$	**Objekttyp**	**Aufwand** E'/min	**Prod.** $q/\text{DDL} \cdot \text{min}^{-1}$
MotionSens	17,05	2,46	OutIllum (x)	312,67	0,37
Occupancy-Control (x)	42,88	3,45	OutTemp-Sens	102,61	0,95
RadiatorAct	6,54	4,74	RadiatiorAct	12,01	3,75
Room (x)	37,14	3,50	Room	107,20	1,39
Temperatu-reControl	97,58	2,70	RoomIllum	133,28	1,43
TempSens	11,05	3,80	RoomTemp	120,11	2,07
			TempSens	14,33	3,21
			Wall	42,90	1,77
			ohne (x)	$\sum = 1001$	Med = 2,14

Vergleich beider Fallstudien

Die beiden vorhergehenden Abschnitte leisteten eine Konsolidierung der Messdaten der Fallstudien *FloorAutomationX* und *RoomAutomationX*. Auf der Basis der dort ermittelten Produktivitätszahlen (q) lässt sich nun ein Vergleich bzgl. der Verbesserung durch Einführung der Automatisierung vornehmen.

Um eine relevante Prozessverbesserung durch die Automatisierung in der *RoomAutomationX*-Fallstudie nachzuweisen, gehen wir – wie in Abschnitt 10.2.1 bereits erwähnt – von der Nullhypothese aus, dass die Automatisierung keine statistisch relevante Prozessverbesserung bringt. Um diese Nullhypothese zu widerlegen müssen wir nun also zeigen, dass die in *RoomAutomationX* gegenüber *FloorAutomationX* gemessenen Produktivitätsgewinne nicht das Ergebnis des Zufalls sind.

Dazu betrachten wir uns zunächst den Box-Plot der Ergebnisse in Abb. 10-9. Auch hier gehen wir zunächst wieder getrennt für die HTML- und die SDL-Dokumente vor und beginnen mit den HTML-Artefakten. Wie man an Hand des Diagramms in Abb. 10-9 bereits erkennt zeigt die *RoomAutomationX*-Fallstudie eine jeweils deutlich höhere Produktivität.

Wir wollen diesen Produktivitätsgewinn statistisch untermauern und setzen dazu die Techniken aus Abschnitt 10.2.1 ein. Zunächst nutzen wir den Wilcoxon-Mann-Whitney-Rangsummentest, um mit statistischer Signifikanz zu zeigen, dass die Verbesserung nicht Produkt des Zufalls ist. Dabei erhalten wir als p-Wert bei exaktem Test 0,0084. Wir können damit also die Nullhypothese, dass es keinen Unterschied

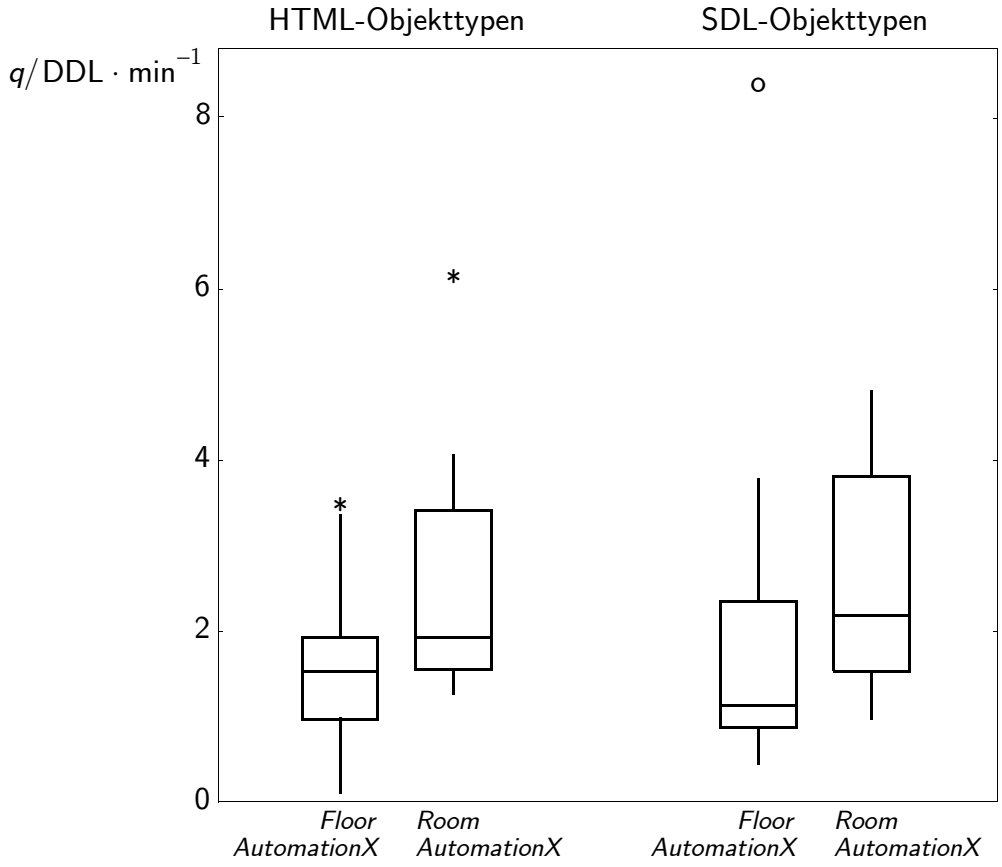


Abbildung 10-9. Box-Plot der Qualitäts-Produktivitäten der Fallstudien

zwischen den HTML-Daten aus *FloorAutomationX* und *RoomAutomationX* gibt, bei einem Konfidenzniveau von $\alpha = 0{,}01$ sicher ablehnen.

Diese signifikante Produktivitätsverbesserung bzgl. der HTML-Artefakte wollen wir mit Hilfe des Hodges-Lehman-Schätzers aus Abschnitt 10.2.1 quantifizieren. Es ergibt sich dabei ein $\hat{\Delta}(q_{\mathrm{HTML}}) = -0{,}72\,\mathrm{DDL/min}$ innerhalb des 95%-Konfidenzintervalls $[-1{,}34, -0{,}24]$. Legen wir die Punktschätzung $\hat{\Delta}$ zu Grunde, so ergibt sich eine Produktivitätssteigerung von 0,72DDL/min (pro Artefakt), die wir auf eine Automatisierung zurückführen können. Bezogen auf den Median von q_{Floor} entspricht dies einer Steigerung um 48%.

Für die SDL-Dokumente gehen wir analog vor und erhalten zunächst einen p-Wert des Rangsummentests von 0,0006 bei exaktem Test. Damit können wir die Nullhypothese wieder sicher ablehnen und berechnen ein $\hat{\Delta}(q_{\mathrm{SDL}}) = -0{,}93\,\mathrm{DLL/min}$ im 95%-Konfidenzintervall $[-1{,}58, -0{,}54]$. Dies entspricht einer „mittleren" Produktivitätssteigerung von 0,93DDL/min pro Artefakt und bezogen auf den Median $\mathrm{Med}(q_{\mathrm{Floor}})$ einer prozentualen Steigerung von 87%. Diese im Vergleich zu den HTML-Artefakten höhere Steigerung liegt sicherlich darin begründet, dass große Teile der SDL-Dokumente aus deren HTML-Spezifikation erzeugt werden konnten (insbesondere die oben erwähnten Ausreißer sind ein Hinweis auf diese Tatsache).

Betrachtet man die Produktivitätssteigerung über alle Dokumente, so ergibt sich eine signifikante Steigerung von $\hat{\Delta}(q_{\text{ges}}) = -0{,}84\,\text{DDL/min}$ und bezogen auf den Median der Produktivitäten der Artefakte von *FloorAutomationX* eine beachtliche Steigerung der Produktivität pro Artefakt um 54%.

10.2.3 Rentabilität der Automatisierung

Bei den obigen Untersuchungen wurde stets angenommen, dass die Automatisierungswerkzeuge existierten und daher der Aufwand der Erstellung dieser Werkzeuge nicht in die Produktivität eingerechnet. Dies ist natürlich nicht ganz fair, da sich auch der Erstellungsaufwand dieser Werkzeuge quasi als Fixkosten auf die damit erstellten Produkte (oder Projekte) verteilt. Zusätzlich zu den obigen Größen muss man daher auch die Kosten der Erstellung der Automatisierungswerkzeuge und die Anzahl der damit durchgeführten Aktivitäten berücksichtigen. Insofern hängt diese Betrachtung eng mit derjenigen zur Bestimmung der Effizienz einer Wiederverwendung (z.B. im Rahmen eines Produktlinienansatzes) zusammen. Wir hatten in der Einleitung (Kapitel 1) genau eine solche Wiederverwendung als einen der Vorteile der Automatisierung angeführt.

Ob sich die zusätzlichen Kosten der Erstellung der Automatisierungswerkzeuge rentieren lässt sich durch eine sog. *Break-Even-Analyse* untersuchen, welche typischerweise in der Betriebswirtschaftslehre ihre Anwendung findet (vgl. z.B. [KoB93, S.516]). Damit wird untersucht, welche Menge eines Produkts verkauft werden müsste, um mit dem Gewinn dessen Fixkosten zu decken („Break-Even-Point“ [KoB93, S.705f.]).

Übertragen auf die Software-Entwicklung ist also zu ermitteln, wie viele Artefakte a mit Hilfe der Automatisierungswerkzeuge entwickelt werden müssten, so dass die Summe der jeweiligen Aufwandsgewinne $\Delta E(a) = E_{\text{man}}(a) - E_{\text{auto}}(a)$ den Aufwand zur Erstellung der Werkzeuge E_{tool} übersteigt. Am „Break-Even“-Punkt muss also gelten:

$$E_{\text{tool}} = \sum_{a} \Delta E(a) \qquad \textbf{(Gleichung 10-6)}$$

Für eine Abschätzung ist diese Darstellung nicht sonderlich geeignet, da die einzelnen Artefakte unterschiedliche Größen aufweisen können und dadurch zusätzlich $S(a)$ als weitere Schätzgröße hinzukommt.

Wir wollen die Break-Even-Analyse feingranularer durchführen und untersuchen daher, ab welcher Gesamtgröße S_{BEP} aller Artefakte sich die Werkzeugerstellung bezahlt machen würde. Demzufolge wird Gl. 10-6 wie folgt umformuliert:

$$E_{\text{tool}} = \Delta e \cdot S_{\text{BEP}}$$

Dabei stellt Δe den „mittleren“ Aufwandsgewinn pro Größeneinheit dar, den man durch Einsatz der Automatisierung erzielen kann. Kennt man Δe und E_{tool}, so lässt sich damit S_{BEP} leicht bestimmen.

Um zu entscheiden, ob man die Automatisierungswerkzeuge nun überhaupt entwickeln soll, müssen Δe und E_{tool} geschätzt werden. Diese Abschätzung lässt sich allerdings nicht zuverlässig durchführen, wie auch Czarnecki et al. in [CzE00, S. 15f.] erläutern. In unserem Fall können wir jedoch eine „a-posteriori“-Rechnung auf der Basis der gemessenen Daten durchführen und damit beurteilen, ob ein solche Automatisierung unter ähnlichen Umständen sinnvoll wäre. Die Bestimmung der Kenngrößen Δe und E_{tool} werden wir in den folgenden Abschnitten vornehmen.

Aufwandsgewinn durch Automatisierung

Setzen wir $e = 1/q$, so können wir wieder den Hodges-Lehman-Schätzer verwenden um den „mittleren“ Aufwandsgewinn pro Größeneinheit zu bestimmen. Es ergibt sich ein $\hat{\Delta}(e) = -0{,}37\,\text{min/DDL}$, was unserem gesuchten Δe entspricht.

Ein $\Delta e = -0{,}37\,\text{min/DDL}$ bedeutet, dass die Entwickler in der Fallstudie mit Automatisierung (*RoomAutomationX*) im Mittel 0,37 Minuten weniger Aufwand zur Spezifikation einer Größeneinheit (DDL) der Artefakte benötigten. Wäre eine Automatisierung bereits für *FloorAutomationX* verfügbar gewesen, so hätte man bei einer Gesamtgröße von 47465 DDL einen Aufwand von 17562 Minuten einsparen können. Dies entspricht einem Aufwand von 1,93 Personenmonaten, wenn man 152 Arbeitsstunden als einen Personenmonat (PM) ansetzt [She93, S. 258]. Der tatsächliche Aufwand von *FloorAutomationX* lag bei 41 485 Minuten oder 4,5 PM (korrigiert um die Variantenbildung und ohne den Aufwand der Ausreißer, vgl Abschnitt 10.2.2).

Erstellungsaufwand der Automatisierungswerkzeuge

So wie man den Aufwand der Erstellung der PROBAnD-Dokumente der Fallstudien gemessen hat (vgl. Abschnitt 10.1.2 auf Seite 307), hätte man auch die Zeiten der Werkzeugerstellung messen können. Diese Zeiten wären allerdings nicht sinnvoll gewesen, da ein nicht unerheblicher Teil des Aufwandes in die Konzeption der in dieser Arbeit vorgestellten Automatisierungstechniken und deren Anwendung und Evaluation floss.

Auch den Aufwand der generischen Teile der Werkzeuge muss man nicht in die obige Beurteilung miteinrechnen, da dieser Aufwand im Rahmen dieser Arbeit geleistet wurde und bei der Anwendung der Technik nicht nochmals anfällt. Konsequenterweise dürfen die Aufwände zur Erstellung von Marshaller/Unmarshaller und dem Browser nicht berücksichtigt werden. Desweiteren können wir davon ausgehen, dass die Syntax-Parser (vgl. Abschnitt 5.1.2 auf Seite 91) für die HTML- und SDL-

Artefakte als externe Bibliothek bereits existierten und nur die jeweiligen produktmodellspezifischen Parse-Vorgänge auf dem damit jeweils erzeugten Syntax-Baum („Tree-Parsing“) verrechnet werden müssen. Zuletzt sollte auch der Aufwand zur Konzeption des Produktmodells nicht einfließen, da dieser Aufwand auch bei einer manuellen Methode zu deren präzisen Definition notwendig ist (siehe dazu [Que02, S. 47f.]).

Berücksichtigt man diese Ausnahmen, dann kann man aus der Größe der zu betrachtenden Werkzeuge auf deren Erstellungsaufwand schließen, also auf den Aufwand, den man investieren müsste, um die Werkzeuge beginnend bei Null zu entwickeln. Die Bestimmung der Werkzeuggröße ist sehr einfach über das NCSS-Maß (siehe Abschnitt 10.1.1 auf Seite 302) möglich. Die Hochrechnung auf einen hypothetischen Entwicklungsaufwand hingegen ist äußerst schwierig.

In der Literatur werden für solche Aufwandsschätzungen, die hauptsächlich der Projektplanung dienen, verschiedene Modelle vorgeschlagen. Das bekannteste Modell ist das *Constructive Cost Model* (*CoCoMo*) von Boehm [She93, S. 258f.]. Die Hauptschwierigkeit bei allen diesen Prognosemodellen ist deren vernünftige Kalibrierung mit Daten vergleichbarer Projekte. Ein für die praktische Anwendung relevantes Ergebnis, das Kemerer et al. in [She93, S. 255ff.] aus der Analyse einer Vielzahl von Fallstudien zogen, ist, dass „Modelle, die in anderen Umgebungen entwickelt wurden, unkalibriert nicht sonderlich gut funktionieren“ [She93, S. 275]. Die Größe des relativen Fehlers der Schätzung lag bei vielen Resultaten im Bereich von 500 – 600%. Da wir für unsere Automatisierungswerkzeuge auf keine relevanten Vergleichsdaten zurückgreifen können, stellt dies ein erhebliches Problem für die Anwendung solcher Modelle dar.

Desweiteren berücksichtigen solche Modelle keine Variationen innerhalb verschiedener Typen von Programmen. So hatten wir im Verlaufe dieser Arbeit an der einen oder anderen Stelle darauf hingewiesen, dass es generische Lösungen gibt, die anspruchsvoller, dafür aber mit weniger Code-Zeilen zu implementieren sind (Beispiel: Browser), oder ob man einfachere aber längere Programme schreiben muss (Beispiel: Parser).

Aus diesen dargelegten Gründen halten wir es für legitim eine Abschätzung auf der Basis unseres Erfahrungswissens und einigen gezielten Messungen durchzuführen. Daher betrachten wir zunächst die Daten einer weiteren, kleinen Fallstudie *Modularisierung*, in welcher ausgehend von einem neuen Produktmodell (als Instanz unsere Produktmetamodells aus Abschnitt 5.1.1 auf Seite 86) ein Satz einfacher Werkzeuge erstellt wurde.

Fallstudie *Modularisierung*

Das Ziel der Fallstudie *Modularisierung*, welche von mir selbst durchgeführt wurde, war die Erstellung einer Werkzeugumgebung zur Analyse und Visualisierung von

Lehrmodulen eines geplanten Bachelor-Studiengangs Informatik. Eingabe waren HTML-Dokumente mit der Beschreibung der angebotenen Module und deren inhaltlichen Abhängigkeiten. In Abb. 10-10 ist das Produktmodell für diese Fallstudie mit seinen vier Objekttypen gezeigt.

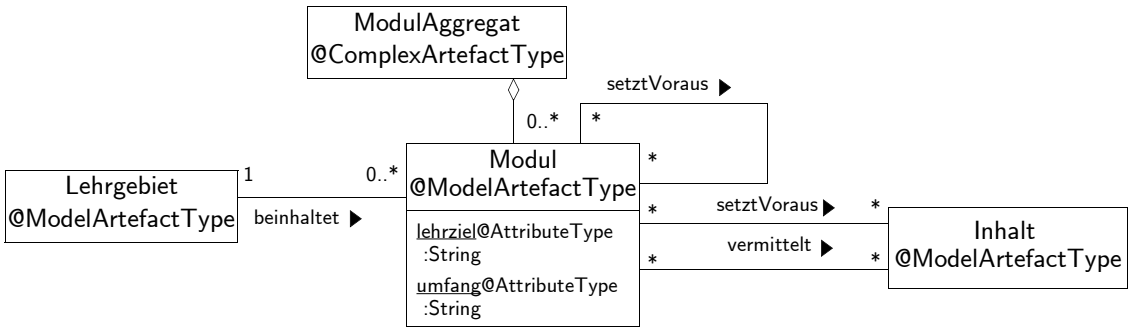


Abbildung 10-10. Produktmodell der Fallstudie *Modularisierung* (abstrakte Artefakttypen)

Sämtliche Module werden in einem ModulAggregat zusammengefasst, welches die Wurzel des Instanzenbaums bildet. Ein Modul ist eindeutig einem Lehrgebiet zugeordnet und setzt andere Inhalte (eines fremden Lehrgebiets) oder andere Module (innerhalb desselben Lehrgebiets) voraus.

Neben einem Parser zur Überführung der konkreten Artefakte der HTML-Dokumente in die abstrakten Artefakte des Produktmodells wurde ein Visualisierungswerkzeug (analog zu Abschnitt 8.4.2 auf Seite 233) erstellt. Desweiteren wurde ein sehr einfacher Analyse-Algorithmus implementiert, der die Abhängigkeiten innerhalb und zwischen Lehrgebieten prüft. Die gemessenen Aufwände und die Größen (NCSS) der Werkzeuge sind in Tabelle 10-11 gezeigt.

Tabelle 10-11. Prozessdaten der Fallstudie *Modularisierung*

Werkzeugerstellung/ Aktivität	**Aufwand** E/min	**Größe** S/NCSS	**relativer Aufwand** $e/\text{min} \cdot \text{NCSS}^{-1}$
Konzeption des Produktmodells	20	—	—
Spezifikation des Produktmodells, Generieren der Basis-Klassen, Erstellung der Code-Rahmen für die Werkzeug-Klassen	50	264 (Basis-Klassen)	0,19
Erstellen des „Tree-Parsers“ für das HTML-Moduldokument (inkl. Konsistenzprüfung der konkreten Artefakte)	160	284 (Werkzeug-Klassen)	0,56
Analyse abstrakter Artefakte (Konsistenzprüfung)	27	18 (Werkzeug-Klassen)	1,5
Implementierung eines Visualisierungswerkzeugs	75	62 (Werkzeug-Klassen)	1,21

Wie zu erwarten war, weist die Erzeugung der Basis-Klassen (inkl. der Spezifikation des Produktmodells) den geringsten relativen Aufwand auf, da sehr viel Code ausgehend von einfachen UML-Modellen erzeugt werden konnte (vgl. auch Abschnitt 5.2.1 auf Seite 97). Das Erstellen der Tree-Parser war aufwändiger, da hier kein Code generiert werden konnte und die hier angesiedelten Konsistenzprüfungen manuell implementiert werden mussten. Dennoch ist die Erstellung noch recht effizient, weil viele ähnliche Stellen (Teile von Tabellen) existieren, deren Code innerhalb des Werkzeugs wiederverwendet werden kann. Die Analyse abstrakter Artefakte und die Erzeugung einer konkreten Repräsentation (wenn auch hier nicht in der Form der eigentlichen Entwicklungsdokumente) war am aufwändigsten. Bei der Analyse kann man das damit begründen, dass ein Großteil der Zeit in die Konzeption der eigentlichen Analysestrategie fließt. Bei der Erzeugung der Visualisierung kann man wieder ein wenig von der Wiederverwendung profitieren.

Auch wenn sich die obigen Aufwandszahlen vernünftig begründen lassen und in unseren Augen auch recht sinnvoll sind, sollte man dennoch die sehr kleine Größe dieser Fallstudie berücksichtigen. Auch die andere Anwendungsdomäne (Informationssystem vs. reaktives System) muss in eine kritische Betrachtung der Zahlen einfließen. Um verlässlichere Daten zu erhalten ist es eventuell angebracht, hier weitere Untersuchungen anzustellen. Insbesondere belegen unsere Erfahrungen, die wir bei der Erstellung der Automatisierungswerkzeuge für die PROBAnD-Methode sammelten, dass die Erstellung des Parsers mindestens genausoviel Aufwand bedeutete wie die Erstellung der Analyse- und Transformationswerkzeuge. Dies liegt vor allem an den aufwändigeren Konsistenzprüfungen, die bei komplexeren Dokumenten wegen der größeren Zahl an Abhängigkeiten notwendig werden. Wenn man überschlagsmäßig die von den PROBAnD-Werkzeugentwicklern aufgebrachte Zeit (in Semesterwochenstunden) in Bezug zu der Größe der Werkzeuge (siehe unten) setzt erhält man deutlich höhere Zahlen (im Mittel $e = 5{,}7$), die man zwar zum Teil auf die oben angesprochenen Änderungen zurückführen kann aber nicht in voller Größe.

Wir werden daher für die folgenden Abschätzungen für alle Werkzeuge von dem schlechtesten Werte aus Tabelle 10-11 ausgehen und diesen als doppelt so hoch annehmen, d. h. wir gehen von $e_{\text{tool}} = 3{,}0$ aus. Auch die Erstellung des Produktmodells und der Basis-Klassen wollen wir doppelt so hoch mit $e_{\text{model}} = 0{,}4$ abschätzen.

Messung der Werkzeuggröße

Die tatsächlichen Größen der Automatisierungswerkzeuge sind in Tabelle 10-12 gezeigt. Ausgehend von diesen Größen wurde der Aufwand der einzelnen Werkzeuge auf der Basis von e_{tool} bzw. e_{model} abgeschätzt. Dabei werden für die folgenden Berechnungen nur die Aufwände der Werkzeuge berücksichtigt, die auch in der Fallstudie *RoomAutomationX* eingesetzt wurden (Parser, Unparser und Checker). Die

nicht verwendeten Tools oder solche, deren generischer Aufwand nicht eingerechnet werden darf (siehe oben), sind mit einem (x) gekennzeichnet.

Tabelle 10-12. Geschätzte Aufwände der Werkzeugerstellung

Werkzeug	**Aktivität**	**Größe** S/NCSS	**geschätzter Aufwand** E/min
–	Spezifikation des Produktmodells, Generieren der Code-Rahmen und Basisklassen	4950	1980
Tree-Parser	HTML	1834	5502
	SDL	1595	5084
Unparser	HTML	1092	3276
	SDL	1558	4674
Checker	Analyse abstrakter Artefakte	679	2036
Analyzer (x)	Feature-Interaktionsdetektion	515	1544
	Graphvisualisierung	287	860
Clipper (x)	Ausschnittsbildung	1150	–
Merger (x)	Zusammenfassung von Teilsystemen	343	–
Browser (x)	Interaktive Tabellendarstellung	971	–
Modifier (x)	Automatische Änderung des Modells	512	–
	ohne (x)	$\sum = 11708$	$\sum = 22552$
	inklusive (x)	$\sum = 14974$	$\sum = 24956$

Ergebnis der Break-Even-Analyse

Jetzt können wir S_{BEP} berechnen, indem wir die in Tabelle 10-12 berechnete Summe der Aufwände als E_{tool} ansetzen und das weiter oben bestimmte Δe in Gl. 10-6 auf Seite 329 einsetzen. Wir erhalten damit ein S_{BEP} von 60951DDL.

Vergleicht man diesen Wert mit der Größe von 47465DDL der Artefakte von Fallstudie *FloorAutomationX* (siehe oben), so hätte sich der Aufwand einer Automatisierung vermutlich bereits im zweiten Projekt einer ähnlichen Größenordnung bezahlt gemacht.

10.2.4 Beitrag der verbesserten Multiebenenmodellierung

Die gesamten Abschätzungen des Erstellungsaufwands der Automatisierungswerkzeuge beruhten auf deren reinen Java-Implementierung. In dieser Arbeit hatten wir jedoch gezeigt, dass sich ein beachtlicher Gewinn bzgl. der Kompaktheit der Algo-

rithmen mit Hilfe unserer Aktionssprache AL++ erzielen lässt. Man könnte daher nun versuchen, den möglichen Beitrag der Sprache AL++ zur weiteren Effizienzsteigerung unseres Automatisierungsansatzes zu quantifizieren.

Leider genügt es für einen korrekten Vergleich nicht, die NCSS-Zeilen der AL++-Lösung derjenigen der Java-Lösung gegenüberzustellen, da wir damit quasi die Größe unterschiedlicher Programmiersprachen vergleichen würden (siehe dazu Abschnitt 10.1.1 auf Seite 302). Auch wenn viele AL++-Ausdrücke nur eine verkürzende Schreibweise für längere Sequenzen von Java-Befehlen sind, gilt dies nicht immer. Insbesondere die Konstrukte zur Multiebenenmodellierung (z. B. die Instanziierung von Relationen) und solche zur generischen Beschreibung (z. B. der `#`-Operator) weisen eine sehr hohe Kompaktheit aus, d. h. man kann mit nur wenig Zeilen AL++-Code sehr mächtige Operationen beschreiben. Bei einer Implementierung mit AL++ muss man folglicherweise davon ausgehen, dass solche Zeilen auch entsprechend mehr Entwicklungsaufwand benötigen.

Die obigen Zahlen hatten bereits die Rentabilität der Java-Lösung gezeigt, weshalb wir zuversichtlich sind, dass die Sprache AL++ hier eine weitere Verbesserung bringen würde, sobald entsprechende Werkzeuge für deren Einsatz zur Verfügung stünden (siehe Abschnitt 11.2.1 auf Seite 343).

10.2.5 Fazit und Diskussion

Durch die quantitative Beurteilung in diesem Abschnitt konnten wir unsere in der Einleitung der Arbeit aufgestellte Hypothese, dass eine effiziente Software-Entwicklung durch den Einsatz effizient erstellter Automatisierungswerkzeuge möglich wird, empirisch belegen. Wir zeigten, dass wir mit den Automatisierungswerkzeugen einen Produktivitätsgewinn von 54% gegenüber einem Prozessdurchlauf ohne Automatisierung erreichen konnten. Die Erstellung der Automatisierungswerkzeuge war sehr effizient in Relation zu dem eingesparten Aufwand, s. d. sich die Erstellung der Werkzeuge bereits bei zwei Projektdurchläufen rentiert hätte.

Es bleibt natürlich eine Restwahrscheinlichkeit, dass alle diese Zahlen nicht den Beitrag der Automatisierung, sondern die Auswirkungen eines anderen Effekts zeigen, der nicht erfasst wurde. Auch die relativ kleine Größe der Fallstudie oder die geringere Anzahl der Entwickler hätte zu einer Verbesserung führen können (da z. B. weniger Kommunikationsbeziehungen existieren). Da neben dem reinen Zahlenwerk allerdings auch viele der Aktivitäten einen offensichtlichen Beitrag leisten – wie wir dies in den vorangegangenen Kapiteln dieser Arbeit zeigten –, sind wir zuversichtlich, dass die gemessenen Ergebnisse den wahren Charakter der Automatisierung widerspiegeln.

10.3 Grenzen und Risiken der Automatisierung

Um zu einer umfassenden Beurteilung des Automatisierungspotenzials zu gelangen, müssen auch die Grenzen und Risiken einer (modellbasierten) Automatisierung diskutiert werden. Neben rein technischen Grenzen (wie z.B. der Umsetzbarkeit) kann eine fehlende (oder falsche) Qualifikation der Prozessbeteiligten hinderlich sein. Desweiteren können einer Automatisierung soziale Überlegungen entgegenstehen.

10.3.1 Technische Aspekte

Das wohl größte Hindernis, das einer Automatisierung innerhalb Entwicklungsprozesses entgegenstehen kann ist, dass dieser nicht über die entsprechende Reife verfügt und daher viele der in einer Automatisierung notwendigen expliziten Modelle (insbesondere das Produktmodell) nicht bekannt oder spezifiziert sind. Auch Grady weist in [Gra92, S.5] darauf hin, dass erst ab einem CMM-Level von 5 („Optimizing“) eine Automatisierung überhaupt erst als „Bedürfnis“ auftaucht.

Das größte Risiko beim Einsatz von Automatisierungswerkzeugen ist, die „Generierung“ von Fehlern, d.h. die Einführung von Fehlern in den erzeugten Dokumenten oder Modellen, die nicht durch den Entwickler entstanden sind.

Es besteht hierbei die Gefahr, dass wirkliche Entwicklerfehler durch eine falsche Art der Automatisierung „verdeckt“ werden. In der PROBAnD-Methode würde eine solche *Verdeckung* wie folgt zustande kommen: Angenommen man entschließt sich die Definition der Attribute, welche die Werte der aktuellen Parameter bei einem Signalempfang speichern, automatisch zu erzeugen. Auf den ersten Blick erscheint dies eine Erleichterung für den Modellierer zu sein, da er nicht mehr diese stupide Definition vornehmen muss. Auch die Automatisierung gelingt leicht, da der Datentyp des aktuellen Parameters über die Relationen des Produktmodells zugänglich ist. In unserem kleinen *RoomAutomation*-Beispiel aus Abschnitt 6.1.3 auf Seite 138 würde man so z.B. das Attribut theta, das die aktuelle Temperatur speichert, die über das Signal newTemp empfangen wird, erzeugen. Was würde aber nun passieren, wenn der Modellierer einen Tippfehler bei der Spezifikation einer weiteren (hypothetischen) Transition machte, also z.B. ^newTemp(eta) anstatt ^newTemp(theta) schreiben würde (vgl. Tabelle 6-5 auf Seite 142)? Interessanterweise erhielte er an keiner Stelle eine Fehlermeldung, da die Variable eta automatisch erzeugt wird und damit die Zuweisung des aktuellen Parameterwerts an diese Variable erfolgt. Zur Laufzeit des Systems würde allerdings ein Fehlverhalten auftreten, wenn der Wert der Temperatur in eta und nicht in theta gespeichert wird, da die Berechnung der Stellgröße die Belegung des Attributs theta als Ausgangspunkt hat.

Auch die Automatisierungs-Tools können Fehler beinhalten, die zu Fehlern in der Spezifikation (den Modellen) führen. Wegen der regelmäßigen Natur solcher

Fehler (sie tauchen in allen Artefakten auf) lassen sich diese jedoch i.d.R. relativ leicht erkennen. Sollte solch ein Fehler allerdings mit einer „Verdeckung" einhergehen, wird dies um einiges schwieriger. Desweiteren werden bei einer automatisierten Durchführung einer Aktivität stets sehr viele Dokumente gleichzeitig bearbeitet. Wird der Fehler entdeckt, so zieht dies ein entsprechendes „Rückgängigmachen" dieser Fehler nach sich, die auch trotz Versionsverwaltung nicht immer ohne Aufwand vonstatten geht. Eine solche Korrektur kann daher eventuell den gesamten Effizienzgewinn zunichte machen.

Diese Risiken des Einführens oder Verdeckens von Fehlern durch Automatisierungswerkzeuge sollten sehr ernst genommen werden und daher bei der Erstellung der Tools große Sorgfalt an den Tag gelegt werden. Insbesondere stellt dies hohe Anforderungen an die Qualifikation der Entwickler und verlangt eine gründliche Arbeitsweise. Wir werden im folgenden Abschnitt daher nochmals auf diesen Aspekt eingehen.

Das Problem der Verdeckung von Fehlern gab schon einen Hinweis darauf, dass man nicht in die Versuchung geraten sollte alles automatisieren zu wollen. Zum einen wird, wenn kaum noch automatisierbare Aktivitäten existieren, der Aufwand einer Automatisierung dieser verbleibenden Aktivitäten deren Nutzen übersteigen. Zum anderen sollte man berücksichtigen, dass auch der Anteil automatisierbarer Aktivitäten begrenzt ist. So nennt Pews Erfahrungswerte, dass nur 40% der Kosten eines Projekts in die Kodierung fließen, 45% der Kosten aber für die Konzeption und die Diskussion über das zukünftige Produkt aufgewendet werden müssen [Pew04]. So ist es offensichtlich, dass Aktivitäten wie die Validierung (vgl. Abschnitt 9.1.1 auf Seite 242) nie von einer Maschine übernommen werden können. Dies gilt prinzipiell für alle kreativen Tätigkeiten. Selbst wir hatten in diesem Abschnitt trotzt der Existenz klar gemessener Größen (NCSS der Werkzeuge) stets das Augenmaß eines Fachmanns eingesetzt (vgl. auch [Wit87]).

Eine weitere technologische Grenze zeichnet sich bei den vertikalen Transformationen ab. Dass die durch das Eingangsmodell (die Spezifikation) und den generierten Code beschriebenen Systeme i.d.R. nicht identisch sein können hatten wir bereits bei der Unterscheidung zwischen Prototyp und Spezifikation in Abschnitt 9.1.2 auf Seite 246 herausgestellt. Die dort gemachten Feststellungen gelten im selben Maße auch für die Generierung von Produkt-Code. Insbesondere bedeutet dies, dass die dort herausgearbeiteten Probleme einer „bedeutungserhaltenden" Abbildung das here Ziel der modellbasierten Entwicklung, sämtliche Code-Artefakte aus Modellen zu erzeugen, gefährden könnten. Im Rahmen der MDA (siehe Abschnitt 5.3.2 auf Seite 120) wird sogar explizit gefordert, dass eine Modelltransformation immer "der Prozess der Umwandlung eines Modells in ein anderes Modell *desselben* Systems sei" [MiM03, S.2-7]. Es wird nach unserer Klassifikation also eine Transformation vom Typ M0 vorausgesetzt. Eine solche Transformation ist allerdings nur in einer idea-

lisierten Situation möglich (vgl. [KlW03]). Es lässt sich in einem praktischen Kontext also nicht unbedingt der gesamte Code des Zielprodukts aus dessen Spezifikation generieren.

Sollte eine solche Generierung aus einer Spezifikation gelingen, müsste man deren Formalismus um entsprechende Konzepte der Implementierungsebene anreichern. Ein sinnvolles Vorgehen wäre daher eine teil-automatisierte (von manuellen Entwicklungsentscheidungen begleitete) Transformation des Analysemodells in ein Entwurfsmodell, welches in dem angesprochenen Formalismus dargestellt werden könnte.

10.3.2 Qualifikationsaspekte

Auch wenn in dem vorliegenden Text stets von Entwicklern, Modellieren oder Spezifizierern gesprochen wurde, treffen diese Rollenbezeichnungen auf das Aufgabenfeld der Automatisierung, so wie sie hier vorgestellt wurde, nicht wirklich zu. Ich glaube, dass hier eine andere (zum Teil höherwertige) Form der Qualifikation notwendig wird.

Dies liegt darin begründet, dass der Abstraktionsgewinn, den man durch die verbesserte Metamodellierung und die Aktionssprache AL++ erzielt, nicht unbedingt von einer Simplifizierung der Tätigkeiten begleitet wird. Hier verhält es sich also anders als bei dem Sprung von Assembler zu den Hochsprachen. Auch wenn durch die Aktionssprache eine Abstraktion von implementierungsnahen Konzepte erfolgt (wie z.B. das Iterieren über eine Menge von Instanzen, vgl. Abschnitt 5.2.2 auf Seite 101), müssen die beteiligten Personen ein sehr genaues Verständnis der (Meta-)modellierung aufweisen. Insbesondere bei der Formulierung generischer Transformationen (wie sie in dieser Arbeit unter anderem in der Form des Marshallers/Unmarshallers in Abschnitt 5.2.4 auf Seite 109 oder des Browsers in Abschnitt 8.4.1 auf Seite 232 vorgestellt wurden) muss eine präzise Kenntniss aller beteiligten Modellebenen vorausgesetzt werden. Darüberhinaus wird bei einer Komposition einzelner automatisierter Aktivitäten zu komplexeren Automatisierungsketten (wie z.B. in Abschnitt 9.2.3 auf Seite 268 bei der Kopplung von reaktivem System und Umgebung oder in Abschnitt 9.3.3 auf Seite 285 beim automatisierten Prototyping) ein sehr genaues Verständnis des Entwicklungsprozesses notwendig. Der aufmerksame Leser dieser Arbeit kann diese Argumentation sicherlich nachvollziehen.

Hier bietet sich meiner Meinung nach die Chance, hoch-qualifizierte Personen für solche Tätigkeiten auszubilden, die es bisher in einer solchen Güte nicht gab. Besonders in der industriellen Praxis kann dieser Mangel ein großes Hindernis für den Einsatz von Automatisierungsansätzen darstellen (siehe dazu auch [Com04]). In einer universitären Informatikausbildung sollten daher grundlegende Fähigkeiten zur Beherrschung von komplexen Systemen und Modellen vermittelt werden. Hier muss

allerdings in Richtung der Fachdidaktik noch einiges unternommen werden. Wie schwierig den Studierenden bereits das Verständnis und die Anwendung etablierter „abstrakter“ Konzepte wie Generalisierung und Design-Patterns fällt, konnten wir deutlich an einem Software-Praktikum im Grundstudium beobachten (siehe dazu den Artikel [Met03b]). Wie die in dieser Arbeit vorgestellten „anspruchsvolleren“ Konzepte angemessen zu vermitteln wären bleibt also zunächst offen. Der Beginn von Kapitel 3 hat desweiteren an Hand der vielen Zitate gezeigt, dass es bereits bei den Grundlagen an einem konsistenten Gesamtbild (vgl. insbesondere [Atk97]) und möglicherweise auch einer (formalen) Theorie der OO-Konzepte fehlt.

Ich vertrete desweiteren die Meinung, dass sich Deutschland – insbesondere in dem in dieser Arbeit behandelten Gebiet der Modellierung und Qualitätssicherung – eine große Chance in der Form einer seiner leider immer stärker vernachlässigten Tugenden böte, der sprichwörtlichen „deutschen Gründlichkeit“ (vgl. [Hen04, S. 12]). Denn wird bei der Formulierung anspruchsvoller Modelle und Automatisierungsalgorithmen nicht sorgfältig vorgegangen, kann dadurch die Qualität des Endprodukts sogar verschlechtert werden. Wir hatten die Gefahr der Einführung von Fehlern weiter oben bereits erläutert.

10.3.3 Soziale Aspekte

Wir hatten bisher angenommen, dass der Aufwandsgewinn, welcher nachweislich durch die Automatisierung erreicht werden konnte, stets zur Sicherung der Produktqualität eingesetzt wird. Es kann natürlich auch eine andere Konsequenz daraus gezogen werden. Da bei einer gleichbleibenden Qualität des Produkts weniger Zeit für dessen Entwicklung aufgebracht werden muss, können die „überschüssigen“ Arbeitskräfte freigesetzt und so die Wertschöpfung des Unternehmens erhöht werden. In anderen Industriebereichen würde man diese Maßnahme als „Rationalisierung“ bezeichnen.

Wie auch dort stellt das Nichtdurchführen solcher Maßnahmen (zum Erhalt der Arbeitsplätze) keine Alternative in einer globalisierten Wirtschaftsgesellschaft dar (vgl. [Rif04, S. 23, 34 und 57]). Denn andere Firmen werden den Wettbewerbsvorteil, der sich durch eine Automatisierung ergibt, nutzen und damit die Konkurrenzfähigkeit der nicht-automatisierten Unternehmen gefährden (und letztlich alle deren Arbeitsplätze). Die traditionelle Argumentation, dass technologischer Fortschritt zwar alte Arbeitsplätze vernichten, im Gegenzug aber mindestens genauso viele neue Jobs schaffen würde, trifft heute nicht mehr zu [Rif04, S. 18f. und 63f.]. Der Zuwachs an Produktivität übersteigt bei weitem den Zuwachs an Nachfrage, der seinerseits die Zahl der Arbeitsplätze bedingt.

Ein Rückgang der Beschäftigungsquote sollte allerdings als Erfolg unserer Gesellschaft gesehen werden, „stupide“ Tätigkeiten, in denen man keine Erfüllung findet,

Maschinen zu überlassen (vgl. [Lie04]). Das Problem, das sich stellt, ist also nicht irgendwelche industriellen oder staatlichen Arbeitsplätze zu schaffen (Stichwort: „ABM“), sondern wie man den Produktivitätsgewinn (und die damit verbundene höhere Wertschöpfung) gerecht auf die Gemeinschaft verteilt und die freigesetzte menschliche Arbeitskraft *sinnvoll* einsetzt.

Eine Lösung, die Rifkin in [Rif04] vorstellt, ist einen „Dritten Sektor“ zu schaffen (oder diesen zu vergrößern). In diesem, auch als unabhängigen oder freiwilligen Sektor bezeichneten, gesellschaftlichem Bereich, „widmet man seinem Mitmenschen Zeit, statt künstliche Marktbeziehungen mit diesen einzugehen und sich und seine Dienste zu verkaufen“ [Rif04, S. 194]. An der erhöhten Wertschöpfung könnte man die Menschen, die im „Dritten Sektor“ tätig sind, z. B. über Steuererleichterungen (falls sie noch einer Tätigkeit in den anderen Sektoren nachgehen) oder über ein „Sozialeinkommen“ (falls sie ohne andere Arbeitsstelle sind) teilhaben lassen (vgl. [Rif04, S. 203ff.]).

Zusammenfassung

Dieses Kapitel beschäftigte sich mit der Beurteilung der Automatisierung indem zunächst der Beitrag einer solchen Automatisierung empirisch untersucht wurde. Der Vergleich zweier Fallstudien aus dem Bereich der Gebäudeautomation lieferte hier den Nachweis einer beträchtlichen Produktivitätssteigerung von 54 %, der durch Einsatz effizient erstellter Automatisierungswerkzeuge erreicht wurde.

Neben diesem Effizienz- bzw. Produktivitätsgewinn wurde die Rentabilität der Automatisierung untersucht und in den berücksichtigten Fallstudien gezeigt, dass sich die Erstellung der Automatisierungswerkzeuge bereits nach zwei Projekten bezahlt gemacht hätte.

Zuletzt wurden die potenziellen Grenzen und Risiken einer Automatisierung in technischer und gesellschaftlicher Hinsicht diskutiert und auf die Chance hingewiesen, die sich durch die Ausbildung hochqualifizierter Fachkräfte im Bereich der Automatisierung bietet.

11 Offene Punkte und Verbesserungsansätze

Where a calculator on the ENIAC is equipped with 18,000 vacuum tubes and weighs 30 tons, computers in the future may have only 1,000 vaccuum tubes and perhaps weigh 1.5 tons.

— [Popular Mechanics, March, 1949]

Auch wenn in dieser Arbeit viele neue Ergebnisse zur modellbasierten Software-Entwicklung und zur automatisierten Qualitätssicherung präsentiert wurden, so wirft doch jede neue Lösung stets weitere interessante Fragestellungen und Probleme auf. An der einen oder anderen Stelle dieser Arbeit wiesen wir bereits auf solche offenen Punkte und mögliche Lösungsansätze oder Verbesserungen hin, die wir nun ausführlicher – gegliedert nach den unterschiedlichen Gebieten dieser Arbeit – behandeln möchten.

11.1 Verbesserte Multiebenenmodellierung

Obwohl wir in dieser Arbeit die verbesserte Multiebenenmodellierung bereits sehr ausführlich betrachtet hatten, bietet sich dennoch Raum für mögliche Ergänzungen.

Eine interessante Fragestellung wird aufgeworfen, wenn man die „Mehrfachvererbung“ zulässt, d.h. dass ein Modellelement mehr als einen Supertyp besitzen kann. Da wir in Abschnitt 3.2.1 auf Seite 40 gezeigt hatten, wie man mit Hilfe einer zusätzlichen Modellebene eine flexible Erzeugung speziellerer Modellelemente durch Instanziierung (und nicht durch Spezialisierung) erreichen kann, muss daher dann über die Konsequenzen oder Möglichkeiten einer „Mehrfachinstanziierung“ nachgedacht werden.

Eine andere mögliche Verbesserung wäre, den Mehraufwand zur Spezifikation „geordneter“ Links zu reduzieren, der durch die Interpretation der Link-Ende als „einfache“ Mengen entsteht. So hatten wir z.B. für die Spezifikation der formalen Parameter eines Signaltyps ein zusätzliches Attribut parameterNbr (siehe Abschnitt B.2.3 im Anhang) und den Constraint $CS2$ (siehe Abschnitt 8.1.2 auf Seite 192) angeben müssen. Einfacher wäre es gewesen, wenn man – wie in der UML – eine `{ordered}`-Annotation hätte vornehmen könnten (in Abschnitt 5.2.1 auf Seite 97 hatten wir dies angedeutet).

Da dieser Fall vermutlich häufiger auftaucht, böte es sich an, stets von einer „geordneten“ Menge auszugehen, also die Reihenfolge, in welcher die Objekte an den Link-Ende eingefügt wurden, beizubehalten. Da dieser Speziallfall auch für alle Anwendungen genügt, in welchem es nicht auf eine solche Reihenfolge ankommt, wäre diese Erweiterung sehr einfach und transparent möglich.

Wie man an Hand der UML sieht, gäbe es noch viele weitere Ergänzungen, die in Spezialfällen eine „komfortablere“ Modellierung erlauben. Ich bin allerdings der Meinung, dass man nicht zu viel von diesem „syntaktischen Zucker“ realisieren sollte, da alle diese Konstrukte von jemandem, der eine solche Sprache lernen möchte, verstanden werden müssen. Interessant ist zu diesem Aspekt z.B. auch der Trend in der SDL-Community (siehe `http://www.sdltaskforce.org/`), wo man momentan versucht, sich auf einen minimalen, sinnvollen Satz von SDL-Modellierungskonstrukten zu einigen.

Allerdings existieren auch mächtigere Modellierungskonzepte wie z.B. die „Assoziations-Klassen“ [RJB99, S.155] oder „Qualifier“ [RJB99, S.398ff.]. Gogolla und Richters zeigen in [GoR99], wie sich diese Konstrukte auf n-äre Relationen und OCL-Constraints abbilden lassen. Da unser Metamodellierungsansatz allerdings nur binäre Relationen unterstützt, wäre an dieser Stelle eine Erweiterung notwendig.

11.2 Aktionssprache AL++

Neben der verbesserten Multiebenenmodellierung, die eine Vervollständigung bzw. Ergänzung der Arbeiten von Atkinson et al. darstellt, ist die Aktionssprache AL++ ein zentraler Teil der vorliegenden Arbeit. Diese Sprache wurde allerdings zunächst nur als hypothetische Sprache (aus „didaktischen“ Gründen der besseren Lesbarkeit und Verständlichkeit) eingeführt. Die *Realisierung* von AL++ als voll funktionsfähige und direkt anwendbare Sprache innerhalb eines entsprechenden Werkzeuges wirft weitere interessante Fragen auf. Wir wollen hier daher auf eine mögliche Realisierung und die eventuellen Erweiterungen von AL++ eingehen.

11.2.1 Realisierung der AL++

Bei einer praktische Umsetzung der AL++ sind zwei Aspekte zu betrachten: Zum einen muss man sich Gedanken über eine sinnvolle maschinelle Repräsentation der Modellartefakte innerhalb eines solchen Werkzeuges machen, zum anderen ist die Darstellung der Modelle und deren Handhabung durch einen Benutzer zu bedenken. Insbesondere muss man sich über eine vernünftige Eingabe von AL++-Code und dessen Ausführung im Klaren werden. Hierzu stehen prinzipiell die Möglichkeit des Kompilierens oder des Interpretierens zur Auswahl, wobei letztere interaktiver und erstere effizienter ist.

Die Interaktivität, also die direkte Ausführung der AL++-Algorithmen, weist den Vorteil sehr kurzer Feedback-Zyklen auf. Damit ließen sich komplexere Transformationen zunächst an einen kleineren Modell erproben und die Konsequenzen für das Modell direkt beobachten. Die Transformation komplexerer Modelle, die einen effizienteren Ansatz benötigen, könnte man dann bei Bedarf nach einer Kompilierung der AL++-Operationen durchführen.

In einer idealen AL++-Entwicklungsumgebung würde man daher sowohl die Transformationsalgorithmen als auch die Modelle nebeneinander bearbeiten können. In Abb. 11-1 ist eine solche Umgebung gezeigt, in welcher sowohl die abstrakten Modellelemente in einem Repository (linke Seite) als auch die AL++-Programme in einem Editor (rechte Seite) bearbeitet werden können.

Die durch eine solche Umgebung ermöglichte Interaktivität und der direkte Zugriff auf das Modell-Repository böten auch die Möglichkeit, die Programmierung in AL++ insofern zu unterstützen, als dass die Ebene und die Potenz der Modellelemente im AL++-Code automatisch eingeblendet werden könnten (dies ist in Abb. 11-1 allerdings nicht gezeigt). Wir hatten eine solche „Einblendung“ z.B. bereits in Abschnitt 4.2.2 auf Seite 65 zur besseren Verständlichkeit der Sprachkonstrukte von AL++ eingesetzt.

In diesem Zusammenhang könnte sich dann auch eine entsprechende Kodierungsrichtlinie zur Groß-/Kleinschreibung der Bezeichner finden lassen. Die von Java (und anderen objektorientierten Programmiersprachen) bekannte Unterscheidung zwischen Instanz und Klasse an Hand der Schreibweise des Anfangsbuchstabens lässt sich leider nicht auf beliebige Metaebenen erweitern. Alternativ zu der Einblendung der Potenzen und der Ebene könnte man hier eventuell auch mit einer sinnvollen Farbkodierung arbeiten. So wäre es z.B. möglich, alle Bezeichner, die eine M_0-Instanz kennzeichnen grün, alle Elemente der Ebene M_1 blau und alle Meta-Elemente (M_2) rot zu färben.

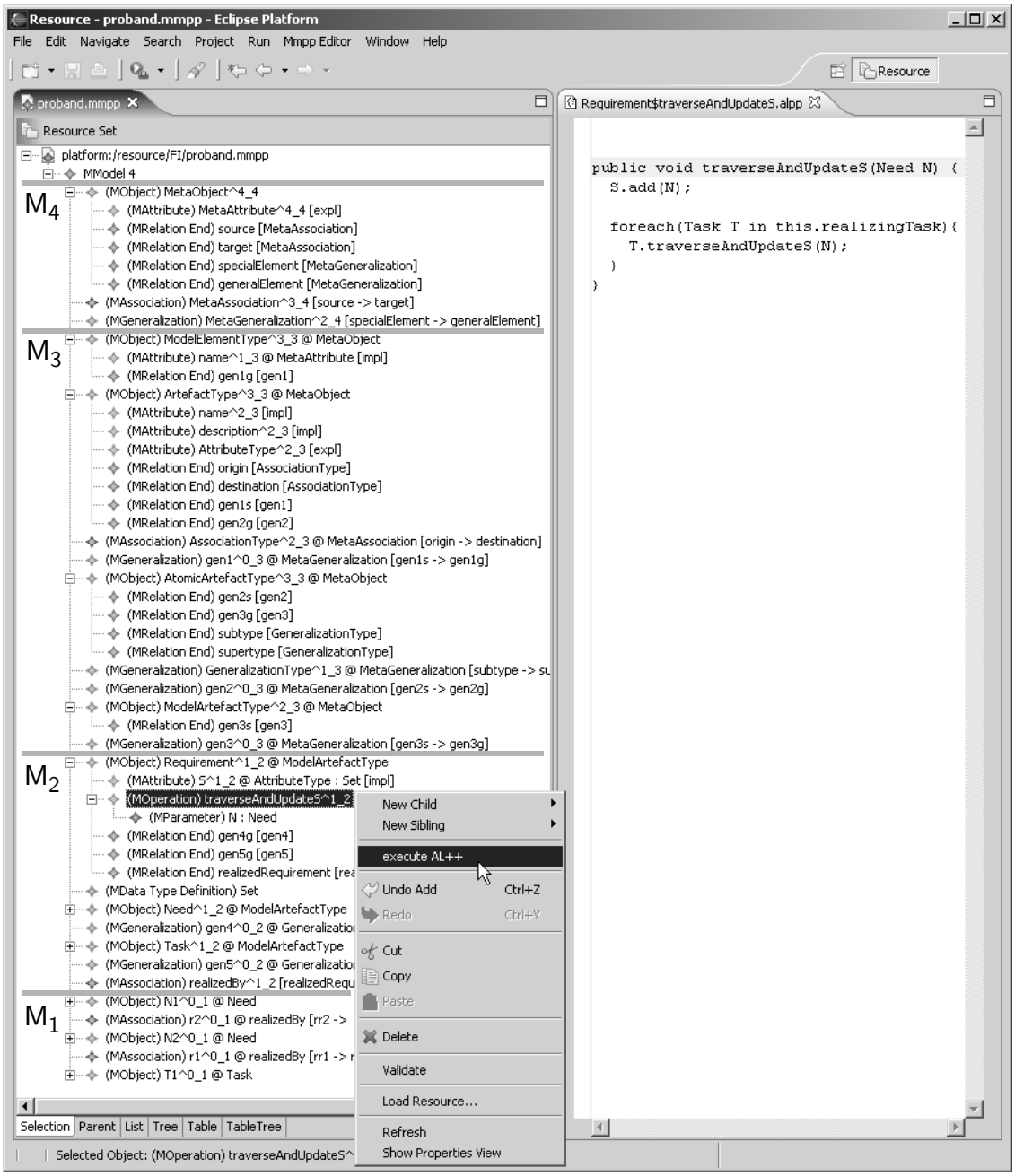


Abbildung 11-1. Beispiel einer möglichen AL++-Entwicklungsumgebung

Viele der zur Umsetzung dieser Konzepte nötigen Techniken können sicherlich aus dem Bereich der *intentionalen Programmierung* (siehe [CzE00, S. 503ff.]) entlehnt werden, da dort ähnliche interaktive Ansätze für die Programmierung zur Anwendung gebracht werden.

Wie bereits angedeutet, sind auf der linken Seite von Abb. 11-1 die abstrakten Modellelemente dargestellt. Die Modellebene eines jeden Elements ist dabei durch ein vorangestelltes „_“, die Potenz durch ein vorangestelltes „^“ gekennzeichnet. Wie man erkennt, beinhaltet dieses Repository Modellelemente der Ebenen M_1 bis M_4

(zur besseren Hervorhebung sind die Modellebenen in der Abbildung nochmals gekennzeichnet).

Die Elemente der Ebene M_4 stellen dabei ein minimales OO-Modell dar, ausgehend von welchem alle anderen Modellelemente instanziiert werden. Für unsere Anwendung ist dieses Modell sehr einfach, es besteht lediglich aus einem Objekttyp MetaObject, der einen explizit instanziierbaren Attributtyp MetaAttribute und zwei Relationstypen (MetaGeneralization und MetaAssociation besitzt).

Die Ebene M_3 entspricht einem Ausschnitt aus dem Produktmetamodell (siehe Anhang B.1) und M_2 ist ein Ausschnitt aus dem Produktmodell der PROBAnD-Methode (siehe Anhang B.2). Schließlich sind auch die Instanzen N1, N2 und T1 der Ebene M_1 dargestellt (sie entsprechen den Elementen des *RoomAutomation*-Beispiels aus Abschnitt 6.1.3 auf Seite 138).

Um die Modellinformation in dem gezeigten Repository abzulegen und werkzeugseitig darauf zugreifen zu können, benötigt man eine entsprechende Datenstruktur. Eine mögliche Konzeption einer solchen Datenstruktur (Datenmodell) ist in Abb. 11-2 skizziert. Dabei ist zu beachten, dass jedes Modellelement, das im Repository abgelegt wird, als eine Instanz eines Objekttyps aus Abb. 11-2 repräsentiert wird. Es besteht somit für jedes Modellelement der Ebene M_n sowohl eine Instanziierungsbeziehung zu einem Typ des obigen Datenmodells als auch zu einem Typ der Modellebene M_{n+1}. Beide Instanziierungsbeziehungen sind also quasi orthogonal zueinander zu betrachten.

Jedes Multiebenenmodell (MModel) besteht aus einer Menge von Modellelementen (MNamedElement), die durch einen Namen gekennzeichnet werden. Zentrales Modellelement ist dabei das MModelElement, welches durch die explizit modellierte Instanziierungsrelation die Beziehungen zwischen MModelElement-Instanzen festhält. So wird z.B. die Instanziierungsbeziehung zwischen N1 in Abb. 11-1 und dem zugehörigen Typ Need durch exakt einen solche Relation beschrieben.

Daneben besitzt MModelElement ein Attribut potency, das die Potenz der jeweiligen Instanz speichert. Anders als die Modellebene, die sich stets durch das Verfolgen der Instanziierungsrelation bis hin zum Typ der höchsten Ebene und damit zum topLevel-Attribut von MModel bestimmen lässt, kann die Potenz für jedes Element von Hand gesetzt werden. Dies ist notwendig, da bei einer Instanziierung die Potenz des Elements auch um mehr als eins verringert werden kann. Wir hatten dies bei der Definition des ModelArtefactTypes des Produktmetamodells bereits ausgenutzt (vgl. Abschnitt 5.1.1 auf Seite 86).

Als elementares Modellelement leiten wir dann MObject von MModelElement ab (das weitere Element MEntity führen wir analog zu Abschnitt 3.2.2 auf Seite 49 ein, um sowohl Objekttypen als auch Datentypen als Typ für ein Attribut zuzulassen). Jedes MObject besitzt dabei eine optionale Menge von Attributen (MAttribute) und

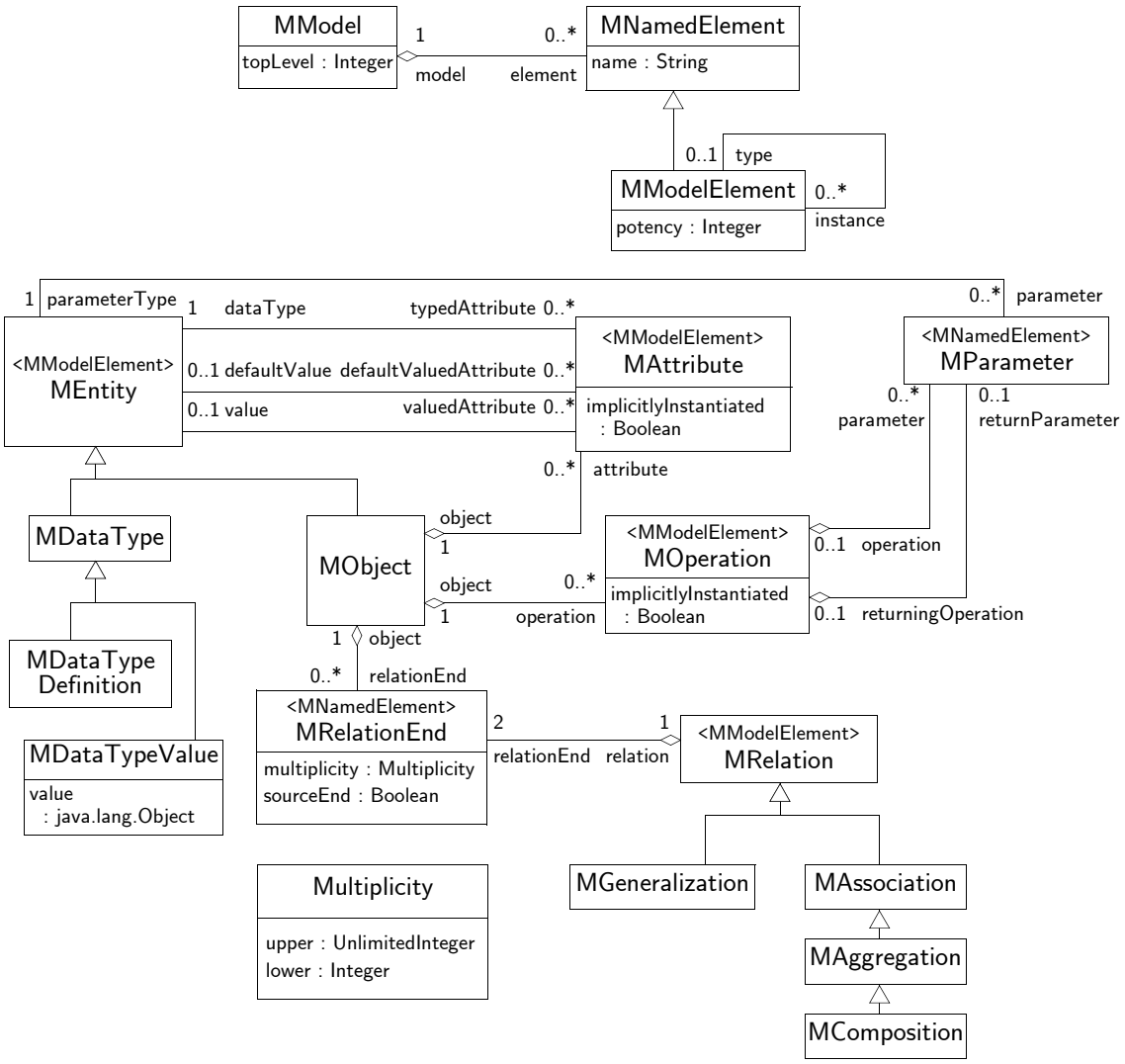


Abbildung 11-2. Mögliches Datenmodell für eine AL++-Entwicklungsumgebung

Operationen (MOperation), wobei bei beiden die Art der Instanziierung (explizit vs. implizit, vgl. Abschnitt 4.1.2 auf Seite 57) im Attribut implicitlyInstantiated festgehalten werden kann.

Daneben können Beziehungen zwischen MObject-Elementen durch Relationen (MRelation) spezifiziert werden. Jede (binäre) Relation besteht aus zwei Relationsenden (MRelationEnd), wobei die Ursprungsseite der Relation durch ein Relationsende, bei welchem das sourceEnd-Attribut auf `true` gesetzt ist, gekennzeichnet wird. Für die Multiplizität des Relationsendes wählen wir den bereits in Abschnitt 4.2.2 auf Seite 65 erläuterten Ansatz analog zur UML.

Der „Datentyp“ eines Attributs wird über die entsprechende Assoziation zu MEntity festgehalten. Daneben werden sowohl eine optionale Anfangsbelegung (defaultValue) als auch die aktuelle Belegung (zur „Laufzeit“) ebenfalls durch Assoziationen zu MEntity spezifiziert. Dies ist möglich, da es sich bei MEntity sowohl um Elemente

der Ebene M_n (der „Datentyp“ des Attributs) als auch um Elemente der Ebene M_{n-1} (die Werte des Attributs) handeln kann.

Da wir als Datentypen auch Java-Klassen zulassen (z.B. das häufig verwendete `Set`, das auch in dem AL++-Code in Abb. 11-1 eingesetzt wird), müssen wir hier eine weitere Verfeinerung des Elements MDataType vornehmen. Zum einen leiten wir MDataTypeDefinition ab, welche den Namen des Datentyps, also z.B. „`Set`“ beinhaltet. Zum anderen definieren wir ein Element MDataTypeValue, welches einen Verweis auf eine Java-Instanz (`java.lang.Object`) trägt.

Zuletzt muss noch MOperation, welche der eigentlichen AL++-Operation entspricht, definiert werden. Hierzu wird ein Attribut body eingeführt, welches den Quelltext trägt. Die formalen Parameter der AL++-Operation werden über die entsprechenden Assoziationen zu MParameter (eine Spezialisierung von MNamedElement) festgehalten. Dabei wird der Rückgabeparameter durch eine separate Relation spezifiziert.

Erste Versuche der oben skizzierten Umsetzung einer AL++-Umgebung liefen bereits erfolgversprechend. Als technische Basis wurde dazu die *Eclipse-Plattform* (siehe `http://www.eclipse.org/`) eingesetzt. Insbesondere konnte mit Hilfe des *Eclipse Modeling Framework EMF* (siehe auch Abschnitt 5.3.3 auf Seite 122) aus dem Modell in Abb. 11-2 ein Satz von Java-Klassen mit den notwendigen Zugriffsmethoden generiert werden, welche das Datenmodell implementieren. Dieser Schritt ist analog zu unserer technischen Realisierung der Automatisierungswerkzeuge wie er in Abschnitt 5.2 auf Seite 96 beschrieben wurde bzw. entspricht dem JMI-Ansatz der Meta Object Facility (vgl. Abschnitt 5.3.1 auf Seite 118). Allerdings generiert das EMF neben den reinen „Modell“-Klassen zusätzlich auch Code zur Darstellung und Modifikation der Instanzen dieser „Modell“-Klassen. Das in Abb. 11-1 gezeigte Repository konnte daher fast vollständig automatisch erzeugt werden (lediglich die spezifischen Bezeichner der Modellelemente wurden angepasst). Besonders vorteilhaft bei der Verwendung des EMF ist, dass alle Modelländerungen auf der Basis eines ausgeklügelten Editor-Mechanismus wieder rückgängig gemacht werden können (siehe z.B. „Undo Add“-Kontextmenü in Abb. 11-1) und somit Fehler in einem Modell, die auf Grund einer fälschlichen AL++-Implementierung entstanden sind, sehr einfach behoben werden können.

Die dynamische Ausführung von AL++-Operationen (siehe „execute AL++“-Kontextmenü in Abb. 11-1) wurde mittels der *DynamicJava*-Bibliothek (siehe `http://koala.ilog.fr/djava/`) umgesetzt. Hierbei handelt es sich um einen Java-Interpreter, der selbst in Java implementiert wurde und daher sehr einfach in existierende Java-Programme eingebunden werden kann. Zu Evaluationszwecken erfolgte zunächst eine manuelle Abbildung der AL++-Konstrukte auf entsprechende Java-Befehle, so wie sie in Abschnitt 5.2.2 auf Seite 101 erläutert wurde. Dieser

Java-Code unterscheidet sich lediglich bzgl. der Spezifika der EMF-Zugriffsmethoden.

Die Realisierung eines AL++-Parsers, der als Eingangsstufe zu einer automatischen Abbildung der AL++-Konstrukte dient, wurde bisher nicht angegangen.

11.2.2 Erweiterung der AL++

Für die in dieser Arbeit vorgestellten Algorithmen waren die Konstrukte der AL++ ausreichend. Es bieten sich allerdings einige Erweiterungen an, welche eine Ausweitung des Einsatzgebiets der Aktionssprache ermöglichen würden.

Zugriff auf weitergehende Elementinformationen

Der programmatische Zugriff auf die Potenz, Modellebene oder Form der Instanziierung (implizit/explizit) eines Elements kann für die Formulierung generischer Algorithmen relevant sein. So dürfte man z.B. nur einem Attribut der Potenz 0 einen Wert zuweisen oder einen expliziten Attributtyp der Potenz 2 weiter instanziieren. Im Datenmodell aus Abb. 11-2 sind diese Informationen bereits spezifiziert, es müssten hier also nur die entsprechenden Sprachkonstrukte eingeführt werden.

Desweiteren müsste man die AL++ insofern erweitern, als dass die Potenz einer Instanz optional angegeben werden kann, da die Beziehung in Gl. 4-1 auf Seite 57 lediglich ausdrückt, dass die Potenz p' des instanziierten Typs T^p kleiner als p sein muss und nicht $p' = p - 1$ gelten muss.

Vollständigerweise müsste man bei der Instanziierung eines expliziten Attributtyps noch angeben, ob dessen Instanzen implizite oder explizite Attributtypen werden sollen (bei einem Attributtyp der Potenz größer 2 ist beides möglich).

Integration deklarativer Ansätze

Wir hatten in Abschnitt 4.2.1 auf Seite 62 diskutiert, dass operationale Ansätze zur Beschreibung der Modelltransformation nicht nur Vorteile mit sich bringen. Eine nutzbringende Verbesserung der AL++ könnte daher die Integration deklarativer Anteile sein.

Für einfache Abfragen und die Spezifikation simpler „Wellformedness“-Regeln würden sich beispielsweise die Konzepte der in Abschnitt 8.1.3 auf Seite 194 vorgestellten Object Constraint Language OCL besser eignen als eine operationale Beschreibung. So ließe sich z.B. mit einem einzelnen sehr einfachen Constraint schnell eine Liste aller Needs erhalten, die nicht durch einen Task realisiert wurden.

Für simple Transformationen, die sich nur lokal auswirken wie z.B. die systematische Änderung von Attributnamen, könnte man auf Ansätze der Graphtransformation zurückgreifen und die komplexeren Teile der Transformationen mit den ope-

rationalen AL++-Befehlen beschreiben. Da mit Hilfe der Graphtransformation eine solche Abbildung mit Bezug auf die Modellentitäten beschrieben werden kann (vgl. dazu insbesondere [WUG03]), lassen sich daher auch ohne die Kenntnis des jeweiligen Metamodells einfache Transformationen beschreiben. Graphtransformationsansätze, die eine grafische Beschreibung erlauben, könnten eine weitere Vereinfachung der Spezifikation solcher (lokaler) Modelländerungen bieten und somit eine Automatisierung von Entwicklungsaktivitäten auch „Laien“ zugänglich machen. Allerdings sind die grafischen Transformationen ab einer gewissen Komplexität bzw. Größe nicht mehr unbedingt leicht verständlich.

Als zukünftige Erweiterung von AL++ könnte also eine Kombination von operationalen und deklarativen Aspekten sinnvoll sein, wobei simple Abfragen und einfache Transformationen mit den deklarativen Teilen, komplexe Transformationen hingegen durch die in dieser Arbeit vorgestellten operationalen Sprachaspekte beschrieben werden könnten.

Erweiterung zur Applikations- und Mehrebenenprogrammierung

Bisher konnten nur die Struktur bzw. nur die im Produktmodell (M_2) definierten Verhaltenskonstrukte durch AL++ transformiert werden. Eine weitergehende Modifikation der Verhaltensaspekte war bisher nicht vorgesehen, da die Sprachelemente (oder Konzepte) von AL++ nicht Teil der M_3- bzw. M_4-Modelle sind.

Solch eine Erweiterung der Metamodelle wäre aber durchaus denkbar. Eine gute Orientierung böte hier die Spezifikation der UML Action Semantics, in welcher die abstrakten Notationselemente zur Beschreibung von Aktionen dargelegt werden. Durch eine entsprechende Anpassung an unsere verbesserte Multiebenenbeschreibung würde das obige Ziel realisierbar.

Damit wäre dann auch eine Meta- bzw. Mehrebenen- oder Mehrstufenprogrammierung, wie wir sie in Abschnitt 5.3.4 auf Seite 126 erläutert hatten, möglich. Inwiefern eine solche Mächtigkeit allerdings noch beherrschbar sein wird bleibt zu klären (für die rein strukturelle Modifikation hatten wir bereits in Abschnitt 10.3 auf Seite 336 Grenzen aufgezeigt).

Umgekehrt kann man die AL++-Konstrukte, die bisher zur Spezifikation von Algorithmen auf den Metaebenen genutzt wurden, auch auf der Modellebene (also M_1) einsetzen. Man erhält somit eine „abstrakte“ oder „very high-level“ Programmiersprache zur effizienten und implementierungsunabhängigen Entwicklung objektorientierter Applikationen (in Abschnitt 4.2.3 auf Seite 75 hatten wir ein kleines Beispiel dazu vorgestellt).

11.3 Automatisierung

Trotz der vielen Anwendungsbeispiele in dieser Arbeit wurden die Möglichkeiten einer Automatisierung noch lange nicht ausgeschöpft. Daher soll hier ein kurzer Abriss über weitere mögliche Anwendungen folgen, von deren Realisierbarkeit ich überzeugt bin.

Übertragung auf andere Entwicklungsmethoden und Domänen

Bereits bei der Definition des Produktmetamodells in Abschnitt 5.1.1 auf Seite 86 wurde auf die breite Anwendbarkeit der Ansätze hingewiesen. So ist es ohne weiteres möglich, alle Methoden, die auf einem Produktmodell fußen, das als Instanz des Produktmetamodells dargestellt werden kann, zu unterstützen. Es sei allerdings nochmals auf die allgemeinen Grenzen der Automatisierung hingewiesen, die wir in Abschnitt 10.3 auf Seite 336 diskutierten.

Spannend wäre es nun, die hier vorgestellten Techniken in einem anderen Kontext zu erproben und dabei auch das Gebiet der reaktiven Systeme, für welches bereits zahlreiche Anwendungen existieren, zu verlassen. Besonders im Bereich der betrieblichen Informationssysteme und dort bei der maschinellen Unterstützung von *Geschäftsprozessen* oder *Workflows* scheint mir das Potenzial einer Automatisierung und der Gewinn durch eine „plattformunabhängige“ Modellierung besonders hoch. Nicht umsonst ist dieses Gebiet das initiale Anwendungsfeld der Model Driven Architecture (vgl. Abschnitt 5.3.2 auf Seite 120) und wird bereits von einigen Werkzeugen (wie z. B. dem ArcStyler von Interactive Objects Software GmbH [Hub02] unterstützt). Auch in diesem Bereich ist die einfache und verständliche Metamodellierung und die Spezifikation mächtiger Transformationen notwendig, weshalb ich sicher bin, dass die in dieser Arbeit vorgestellten Ansätze gewinnbringend eingesetzt werden können.

Eine Problematik, die insbesondere in dem eben genannten Feld der betrieblichen Informationssysteme auftaucht, ist, dass viele Altsysteme (in Form von „Legacy-Code“) existieren, die es in die neue Software-Umgebung zu integrieren gilt. Wie dies sinnvoll mit einem modellbasierten Ansatz zu vereinbaren ist, bleibt zu untersuchen und zu evaluieren. Ansätze wie z. B. das „Reverse-Engineering“ oder auch das „Harvesting“, das von Hubert in [Hub02] vorgestellt wird, bieten hier allerdings bereits erste Lösungsansätze.

Automatisierte Anwendung von Mustern

Eine typische horizontale Entwicklungsaktivität ist die Anwendung von *Mustern*. Traditionell werden solche Muster während des Entwurfs im Sinne der „Design Patterns“ eingesetzt (siehe dazu z. B. das Standardwerk von Gamma et al. [GHJ97]). Das Konzept der Muster lässt sich allerdings auch sinnvoll auf der abstrakten Ebene

der (Analyse-)Modelle einsetzen. So stellen z. B. France et al. in [FGS03] einen musterbasierten Ansatz zur Refaktorisierung von Modellen vor.

Die in der vorliegenden Arbeit vorgestellten Techniken eignen sich in meinen Augen sehr gut für den transformationellen Teil eines solchen Musters. Daneben müsste allerdings – typisch für ein Pattern – auch eine präzise Problem- und Lösungsspezifikation angegeben werden.

Formale Verifikation

Neben der Erzeugung von PROBAnD-Dokumenten (in der HTML- bzw. SDL-Syntax) wurde in dieser Arbeit auch gezeigt, dass sich andere Darstellungsformen (z. B. zur Graphvisualisierung, siehe Abschnitt 8.4.2 auf Seite 233) sehr systematisch und einfach erzeugen lassen.

Ausgehend von der in den abstrakten Artefakten beinhalteten Modellinformation könnte man nun auch Ausgabedokumente erzeugen, die sich als Eingabe in ein Werkzeug zur formalen Verifikation eignen. Varro zeigt in [Var04] eindrucksvoll, wie sich die Überführung eines domänenspezifischen Modells in die Eingabe für einen „Model-Checker" realisieren lässt. Basierend auf dieser Arbeit ließen sich sicherlich analoge Resultate für die Spezifikation reaktiver Systeme erzielen, womit die Palette der Qualitätssicherungsmaßnahmen sinnvoll ergänzt werden würde.

Aspektorientierte Modellierung

Durch die in Abschnitt 6.2.3 auf Seite 151 eingeführte modifizierte Automatenmodellierung wurde es möglich, komplexe Zustandsautomaten innerhalb eines Control-Object-Types durch die Komposition einzelner Transitionen (oder deren Teile) zu beschreiben. Teilmengen der Transitionen unterschiedlicher Objekttypen könnte man nun auch als „*Aspekte*" (also querschneidende Merkmale) interpretieren, womit die Realisierung einer *aspektorientierten Modellierung* (siehe dazu z. B. [FrR04]) sehr einfach möglich wird. Die Instrumentation von Modellen, die wir in Abschnitt 6.3 auf Seite 167 und in Abschnitt 9.3.1 auf Seite 276 betrachteten, hatte einen ähnlichen Charakter.

Alternativ zu der Realisierung der Aspekte durch die Spezifikation von Transformationsalgorithmen ließen sich unterschiedliche Aspekte auch in getrennten PROBAnD-Modellen mit einer identischen Modellarchitektur (Objektstruktur) spezifizieren und diese einzelnen „Aspekt-Modelle" dann zu einem gemeinsamen Modell verweben. Die Realisierung eines solchen „Weaver"-Werkzeuges würde große Ähnlichkeiten zu dem Merger-Tool aus Abschnitt 9.2.3 auf Seite 268 aufweisen.

Benutzerdefinierte Transformationen

Wie diese Arbeit an vielen Stellen zeigte, können manuelle Entwicklungsaktivitäten mit Hilfe der Aktionssprache AL++ operationalisiert und damit automatisiert und effizient zur Ausführung gebracht werden. Dabei ist eine solche Implementierung von Transformationen für ein einzelnes System oftmals bereits weniger aufwändig als die Durchführung der manuellen Tätigkeiten (man denke z.B. an die Instrumentierung eines komplexen reaktiven Systems).

Zusammen mit der in Abschnitt 11.2 auf Seite 342 vorgestellten interaktiven Modellierungs- und Transformationsumgebung auf Basis der verbesserten Multiebenenmodellierung und AL++ lässt sich diese Effizienz meiner Meinung nach weiter steigern. Damit wird eine Implementierung von Transformationen immer attraktiver gegenüber der manuellen Ausführung einer solchen Entwicklungsaktivität. Im Einzelfall bedeutet dies, dass ein Entwickler die Wahl hat, eine Aktivität manuell durchzuführen oder aber – wenn er diesen Aufwand als zu hoch oder die Tätigkeit als zu „langweilig“ einstuft – eine Automatisierung dafür zu realisieren. Auch Sendall und Kozaczynski fordern in [SeK03], dass eine solche benutzerdefinierte Definition von Transformationen möglich sein solle.

Wie eine solche Möglichkeit von Entwicklern in der Praxis angenommen wird muss sich allerdings erst zeigen. Insbesondere die in Abschnitt 10.3.2 auf Seite 338 herausgestellten Qualifikationshürden könnten hier hinderlich sein.

11.4 Modellierung reaktiver Systeme

Mögliche Verbesserungen der PROBAnD-Methode, von welchen auch einer Modellierung reaktiver System im Allgemeinen profitieren könnte, sind während dieser Arbeit bereits an der einen oder anderen Stelle angeklungen. Hier wollen wir zwei wichtige Aspekte hervorheben, die Ergänzung der PROBAnD-Artefakte um Dokumente, die auf einer Szenario-Notation basieren, und die weitere Verfeinerung der Control-Object-Types.

11.4.1 Spezialisierung des Produktmodells

In einigen Situationen waren wir bei unseren Automatisierungsalgorithmen gezwungen, Annahmen über die bearbeiteten Control-Object-Types zu treffen. So nahmen wir in Abschnitt 8.2.1 auf Seite 209 z.B. an, dass alle Blattknoten im Instanziierungsbaum entweder Sensoren oder Aktuatoren waren. Hierfür wäre eine Spezialisierung des ControlObjectTypes in Sensor, Actuator und eventuell andere Objekttypen sinnvoll. Für die Domäne der Gebäudeautomationssysteme wären sicherlich auch Objekttypen wie Controller, Area (mit den weiteren Spezialisierungen Room,

Hallway, etc.) oder Opening (mit den spezielleren Objekttypen Door, Window, usw.) sinnvoll.

Eine solche Verfeinerung könnte man prinzipiell durch die Einführung einer weiteren Modellebene zwischen M_1 (der Spezifikation) und M_2 (dem Produktmodell) realisieren. Zimmermann zeigt in [Zim03] und [ZiM04], wie eine solche Verfeinerung der Produktmodelle in der Domäne der Gebäudesimulation (welche auch die Gebäudeautomation einschließt) aussehen könnte.

Speziell würde eine solche zusätzliche Metaebene weitergehende Analysen der Modelle ermöglichen. In Abschnitt 7.1.1 auf Seite 177 hatten wir auf das Problem hingewiesen, dass trotz unseres PROBAnD-Produktmodells viele Modellelemente semantisch noch nicht „reich“ genug sind. So konnten wir z. B. bisher nicht feststellen, dass eine Gebäudesteuerung sinnlos ist, weil sie eine Helligkeitsregelung in einem Raum (Area) ohne Fenster und Türen (Opening) beschreibt.

Desweiteren ließe sich durch die Einführung solcher Modellelemente die vertikale Modelltransformation (also in Richtung Code) effizienter gestalten. Analog zu den domänenspezifischen Sprachen (siehe Abschnitt 5.3.4 auf Seite 126) wäre es dann möglich spezielle Code-Komponenten z. B. für Sensoren anzulegen oder bei der SDL-Generierung spezielle Verhaltenskonstrukte, welche zur Kommunikation mit der Umgebung des reaktiven Systems benötigt werden, automatisch zu erzeugen.

Eine weitere essenzielle Erweiterung des Produktmodells von PROBAnD wäre die Einführung einer Spezialisierungsrelation zwischen Control-Object-Types. Bisher ist dies nicht vorgesehen, weshalb die Variantenbildung in dieser Methode noch sehr aufwändig ist, da sie quasi nur durch ein „copy and paste“ möglich ist. Insbesondere die sinnvolle Forderung, dass ein Task immer nur von einem Control-Object-Type implementiert werden darf, bedeutet dass nicht nur der Objekttyp sondern auch die von diesem implementierten Tasks „kopiert“ werden müssten. Hier könnte man einer Erweiterung so vornehmen, dass alle vom generellen Control-Object-Type realisierten Tasks auch immer von den spezielleren Objekttypen implementiert werden.

11.4.2 Message/Transition Charts (MTCs)

In Abschnitt 9.1.4 auf Seite 254 hatten wir bereits darauf hingewiesen, dass es in der PROBAnD-Methode noch eine Lücke zwischen den Anforderungen (Needs und Tasks) und deren Implementierung innerhalb der Control-Object-Types in der Form unserer erweiterten Zustandsautomaten gibt. Dort hatten wir weiter erläutert, welche Vor- und Nachteile eine Szenario-Notation in unseren Augen besitzt. Insbesondere lässt sich das Gesamtverhalten des Systems nur schwer verstehen, wenn man dies erst aus der Spezifikation der einzelnen (isoliert beschriebenen) Zustandsautomaten ableiten muss (vgl. [RGG03]).

Eine existierende Notation, welche diese Probleme beseitigt, konnten wir für unsere Anwendungsdomäne bisher nicht identifizieren: So kann man mit UML Aktivitätsdiagrammen zwar sowohl die Interaktion zwischen Komponenten als auch ansatzweise das interne Verhalten beschreiben [Boc04], bei vielen Zuständen wird diese Notation allerdings sehr schnell unübersichtlich. Die von Rößler et al. in [RGG03] vorgestellte CoSDL-Sprache zur Beschreibung von Kollaborationen eignet sich zwar sehr gut für die Domäne der Telekommunikationssysteme (was durch deren Nähe zu SDL zum Ausdruck kommt), die Zustände der einzelnen Objekte spielen hier allerdings eine eher untergeordnete Rolle. Wie auch bei den UML Aktionsdiagrammen lassen sich diese zwar angeben, sind aber nicht zentraler Teil der Diagramme (vgl. [RGG03, S.7ff.]). Desweiteren fällt die Identifikation von Objektgrenzen schwer, da diese nicht explizit grafisch gekennzeichnet werden sondern nur durch die Namen der Objekte entlang der Kollaboration beschrieben sind.

Wir konzipierten daher eine Notation, welche die obige Lücke zu den Szenarien zu schließen vermag und dadurch ein für unsere Anwendungsdomäne besser überschaubares Gesamtbild des Systems vermitteln kann (siehe [ZiM04]). In dieser von uns *Message/Transition Charts* (*MTC*s) genannten Notation können viele Szenarien gleichzeitig und zusammen mit dem internen Verhalten der Objekte beschrieben werden. Dazu werden einige der in Abschnitt 3.1 auf Seite 30 eingeführten grundlegenden Modellierungskonzepte verwendet. In Abb. 11-3 ist zunächst ein Beispiel unserer Notation gezeigt, an welchem wir diese Konzepte erläutern wollen (es handelt sich hierbei einen Ausschnitt aus dem *RoomAutomation*-System von Abschnitt 6.1.3 auf Seite 138).

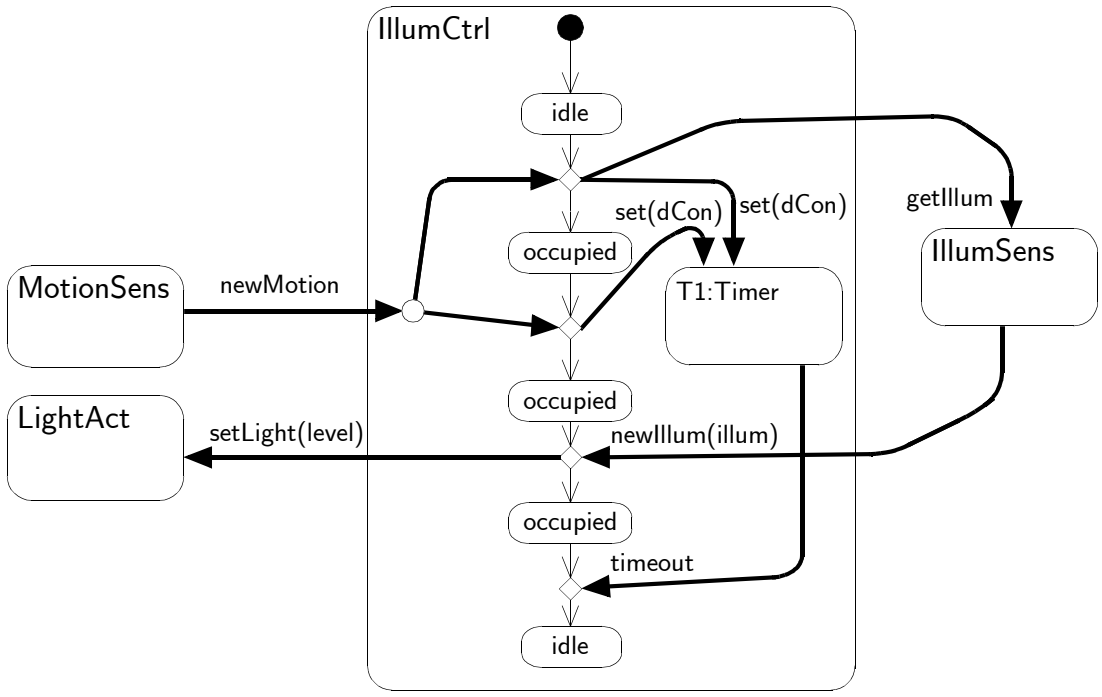


Abbildung 11-3. Beispiel für ein einfaches MTC

Ein abgerundetes Rechtecke beschreibt eine *Objektgrenze*. Innerhalb einer solchen Grenze können die *Zustände* des jeweiligen Objekts durch kleine Ovale gezeichnet werden. Zur besseren Lesbarkeit ist eine Wiederholung gleicher Zustände erlaubt. Zwischen diesen Zuständen werden die *Transitionen* durch einen dünnen Pfeil mit einer Raute gekennzeichnet. Zu Beginn der Spezifikation können die Objekte zunächst auch ohne Zustände und Transitionen modelliert werden (Beispiel: MotionSens in Abb. 11-3).

Auslöser der Transitionen sind *Nachrichten*, was durch dicke Pfeile, die am Rautensymbol der Transition enden, gekennzeichnet wird. Hierbei können solche Nachrichten sowohl ein Ereignis darstellen (im Beispiel die Nachricht newMotion) oder aber auch eine Delegation von Aufgaben (im Beispiel die Nachricht getIllum) modellieren. In Abschnitt 3.1.2 auf Seite 35 hatten wir auf eine Verwendung von Nachrichten für diese beiden Zwecke bereits hingewiesen. Sobald eine Transition durch eine solche Nachricht ausgelöst wird, wird als Aktion typischerweise wieder eine weitere Nachricht erzeugt. Als Einschränkung können diese Transitionen (wie von den Statecharts bekannt) mit Guards versehen werden (vgl. Abschnitt 6.2.3 auf Seite 151). Desweiteren können die Nachrichtenflüsse an den durch kleine Kreise markierten Verbindungspunkten zusammengeführt und wieder auseinandergenommen werden. Im Beispiel wurde dies für die Nachricht newMotion getan.

Solche zunächst flachen MTCs können durch die Spezifikation von Typen von Objekten (Objekttypen) hierarchisiert werden. Bei der Spezifikation solcher Objekttypen werden ausgezeichnete Nachrichtenverbindungspunkte (ähnlich den Gates bei SDL [OFM94, S. 133] oder Ports bei den UML Architektur- bzw. Kompositionsstrukturdiagrammen [JRH03, S. 123ff.]) definiert, mit welchen die Objekte (Instanzen der Objekttypen) zu komplexeren MTCs zusammengesetzt werden können. Die Verbindung der Objekte erfolgt über Nachrichten, deren Namen den Namen der Verbindungspunkte entsprechen müssen. In Abb. 11-3 ist die Anwendung einer solchen Hierarchisierung für den Timer T1 gezeigt. Dieser ist eine Instanz des Typs Timer, der die Nachrichtenverbindungspunkte set, timeout und cancel (letzterer wird im Beispiel nicht verwendet) besitzt.

Erste Anwendungen dieser Methode zur Spezifikation von Gebäudesimulatoren, welche die Wechselwirkungen von Personenaktivitäten berücksichtigen, waren erfolgreich (siehe [ZiM04]), weshalb wir zuversichtlich sind, dass sich eine solche Notation auch in dem größeren Bereich der reaktiven Systeme bewähren kann.

12 Zusammenfassung

In der vorliegenden Arbeit wurde gezeigt, wie durch eine Automatisierung von Software-Entwicklungsaktivitäten ein Effizienz- und Qualitätsgewinn und damit die Beherrschbarkeit komplexer Aktivitäten erreicht werden kann. Damit wurden die in der Einleitung zu dieser Arbeit formulierten Ziele erreicht.

Neue wissenschaftliche Ergebnisse wurden hierbei sowohl für die Grundlagen einer modellbasierten Entwicklung als auch für die Anwendung der modellbasierten Techniken für die Automatisierung von Entwicklungsaktivitäten erzielt.

Die wichtigsten Resultate der ersten Kategorie sind:

1. Die *verbesserte Multiebenenmodellierung*, die anders als die „traditionelle“ Metamodellierung (wie sie z.B. zur Definition der UML verwendet wird), die explizite Spezifikation der Instanziierbarkeit aller Modellelemente (Objekttypen, Relationen, Attribute und Operationen) erlaubt und damit eine deutliche Vereinfachung und bessere Verständlichkeit der Metamodelle ermöglicht (Kapitel 4, Abschnitt 1).
2. Die *Aktionssprache AL++* zur operationalen Beschreibung von Modelltransformationen im Kontext der verbesserten Multiebenenmodellierung. Durch die Einbettung der Sprache in die verbesserte Multiebenenmodellierung wird insbesondere eine natürliche Realisierung von Reflexionsmechanismen möglich und dadurch die Beschreibung generischer Algorithmen deutlich vereinfacht (Kapitel 4, Abschnitt 2).
3. Die *modifizierte Automatenmodellierung*, bei welcher erweiterte Endliche Automaten als Komposition einzelner, getrennt modellierter Zustandsübergänge (Transitionen) spezifiziert werden können. Dadurch wird eine feingranulare und eindeutige Verfolgbarkeit zu den Anforderungen und ein Beherrschen der Komplexität möglich, da man sich jeweils auf relevante Ausschnitte aus einem mitunter komplexen Automaten konzentrieren kann (Kapitel 6, Abschnitt 2.3).

Bezüglich der Anwendung dieser Techniken für die Qualitätssicherung durch Automatisierung wurden die folgenden relevanten Ergebnisse erzielt:

1. Die automatische *Detektion von Feature-Interaktionen* (also kritischer Wechselwirkungen zwischen Produktmerkmalen), die in dieser Form erstmalig für den Bereich der reaktiven Systeme durchgeführt wurde, ermöglichte die automatische Identifikation kritischer Fehler in den Spezifikationen von Gebäudeautomationssystemen (Kapitel 8).
2. Auf der Basis der Generierung von Prototypen aus Anforderungsspezifikationen, der vollständig automatisierten Ausführung solcher Prototypen, der Evaluierung der Prototyp-Läufe und schließlich der maschinellen Modifikation der Spezifikationen konnten Software-Entwicklungsexperimente in einem „*virtuellen Labor*“ durchgeführt werden. Als wichtigstes experimentelles Ergebnis wurde nachgewiesen, dass eine statische Parametrisierung eines Gebäudeautomationssystems sinnvoll durchgeführt und damit eine Reduktion der notwendigen Produktmerkmale erreicht werden kann (Kapitel 9).
3. Die Erzeugung von Entwicklungsdokumenten und die konsistente Änderung vorhergehender Dokumente wurde vollständig durch Werkzeuge realisiert. Durch eine solche Werkzeugunterstützung konnte in einer Fallstudie ein *Produktivitätsgewinn* von 54% gegenüber einer Fallstudie ohne eine solche Werkzeugunterstützung erreicht werden. Die Erstellung der Werkzeuge hätte sich dabei bereits nach zwei Entwicklungsprojekten bezahlt gemacht (Kapitel 10).

— Anhänge —

Anhang A: Die Aktionssprache AL++

Die folgenden Ausdrücke beschreiben die Operationen der Aktionssprache AL++ (Action Language ++). Eine ausführliche Erklärung der Konstrukte findet sich in Abschnitt 4.2.2 auf Seite 65. Es wird das folgende Namensschema verwendet:

- `<referenz>`: Dies entspricht einem Verweis auf eine Entität, um auf diese mit den Mitteln der Sprache zugreifen zu können; so z.B. mit einer Objektreferenz auf die entsprechende Instanz eines Objekttyps; erfolgt keine Zuweisung an `<referenz>`, dann darf hier auch immer ein `<bezeichner>` verwendet werden, wie z.B. bei `new Dreieck()`.
- `<name>`: Hierbei handelt es sich immer um einen String, welcher stets in „ " “ eingeschlossen wird.
- `<bezeichner>`: Ein Bezeichner dient der eindeutigen Identifkation eines Modellelementes, wie z.B. eines Attributs. Hierbei handelt es sich nicht um einen String sondern um reinen Text, wie z.B. in `this.name`.

Desweiteren gibt $\mathrm{M}_{v/\mathrm{d}}$ jeweils die Verwendungsebene (M_v) und Definitionsebene (M_d) eines Elements an.

A.1 Handhabung von Objekten und Typen

Erzeugung von Objekten (new-Operator)

`<`M_n`-ObjektReferenz> = new <`M_{n+1}`-ObjektReferenz>(<`M_n`-ObjektName>);`

Löschen von Objekten

Erfolgt automatisch, falls keine Referenz auf das Objekt mehr existiert.

Bestimmen des Objekttyps (type-Operator)

`<`M_{n+1}`-ObjektReferenz> = <`M_n`-ObjektReferenz>.type;`

Gedachter Supertyp (anyinstance-Operator)

```
<M_{n+2}-ObjektReferenz>.anyinstance <M_n-ObjektReferenz>
```

A.2 Handhabung von Attributen

A.2.1 „Normale“ Attribute

Die folgenden Konstrukte sind für Attribute einer Potenz $p \leq 1$ anwendbar.

Wertzuweisung

```
<M_n-ObjektReferenz>.<M_{n/n+1}-AttributBezeichner> = <attributWert>;
```

Auslesen von Attributwerten

```
<variablenReferenz> = <M_n-ObjektReferenz>.
 <M_{n/n+1}-AttributBezeichner>;
```

A.2.2 Explizit instanziierte Attribute

Die folgenden Konstrukte sind für Attribute einer Potenz $p > 1$ anwendbar, welche explizit instanzziert werden. Die implizite Instanziierung von Attributen erfolgt ohne „Benutzereingriff“.

Erzeugen von Attributen

```
<M_n-ObjektReferenz>.<M_{n/n+1}-AttributTypBezeichner> +=
 (<M_{n-1/n}-AttributName>, <M_n-DatenTypReferenz>);
```

Bestimmen aller Attributnamen

```
<mengenReferenz> = <M_n-ObjektReferenz>.
 <M_{n/n+1}-AttributTypBezeichner>;
```

Bestimmen des Datentyps eines Attributs

```
<M_n-DatentypReferenz> =
 (<M_n-ObjektReferenz>.<M_{n/n+1}-AttributTypBezeichner>,
 <M_{n-1/n}-AttributName>);
```

A.3 Handhabung von Relationen

A.3.3 „Normale“ Relationen

Die folgenden Konstrukte sind für Relationen einer Potenz $p \leq 1$ anwendbar.

Link-Erzeugung

Bei einer Multiplizität von höchstens „1“ am Ziel mit:

```
<M_n-ObjektReferenzUrsprung>.<M_n/n+1-RollenBezeichner> =
 <M_n-ObjektReferenzZiel>;
```

ansonsten mit:

```
<M_n-ObjektReferenzUrsprung>.<M_n/n+1-RollenBezeichner> +=
 <M_n-ObjektReferenzZiel>;
```

Zugriff auf Link-Enden

Bei einer Multiplizität von höchstens „1“ am Ziel mit:

```
<M_n-ObjektReferenzZiel> =
 <M_n-ObjektReferenzUrsprung>.<M_n/n+1-RollenBezeichner>;
```

ansonsten mit:

```
<mengenReferenz> = <M_n-ObjektReferenzUrsprung>.
 <M_n/n+1-RollenBezeichner>;
```

A.3.4 Höherpotente Relationen

Die folgenden Konstrukte sind für Relationen einer Potenz $p > 1$ anwendbar.

Relationserzeugung

```
<M_n-ObjektReferenzUrsprung>.<M_n/n+1-RollenBezeichner> +=
 <M_n-ObjektReferenzZiel>, (<M_n-1/n-RelationsName>,
 <M_n-1/n-MultiplizitaetUrsprung>, <M_n-1/n-MultiplizitaetZiel>,
 <M_n-1/n-RollenNameUrsprung>, <M_n-1/n-RollenNameZiel>);
```

Zugriff auf Relationsinformation

```
(<M_n-1/n-RelationsName>, <M_n-1/n-MultiplizitaetUrsprung>,
 <M_n-1/n-MultiplizitaetZiel>, <M_n-1/n-RollenNameUrsprung>,
 <M_n-ObjektReferenzZiel>) =
 (<M_n-ObjektReferenzUrsprung>.<M_n/n+1-RelationsBezeichner>,
 <M_n-1/n-RollenNameZiel>);
```

Bestimmen aller Rollennamen

```
<mengenReferenz> = <M_n-ObjektReferenzUrsprung>.
 <M_n/n+1-RelationsBezeichner>;
```

Bestimmen aller Rollennamen in einer Richtung

```
<mengenReferenz> = <M_n-ObjektReferenzUrsprung>.
 <M_n/n+1-RollenBezeichner>;
```

A.4 Sonstige Operatoren

Iterieren über Mengen (foreach-Operator)

```
foreach(<objektReferenz> in <mengenReferenz>)
    <anweisungsBlock>
```

Generischer Zugriff (#-Operator)

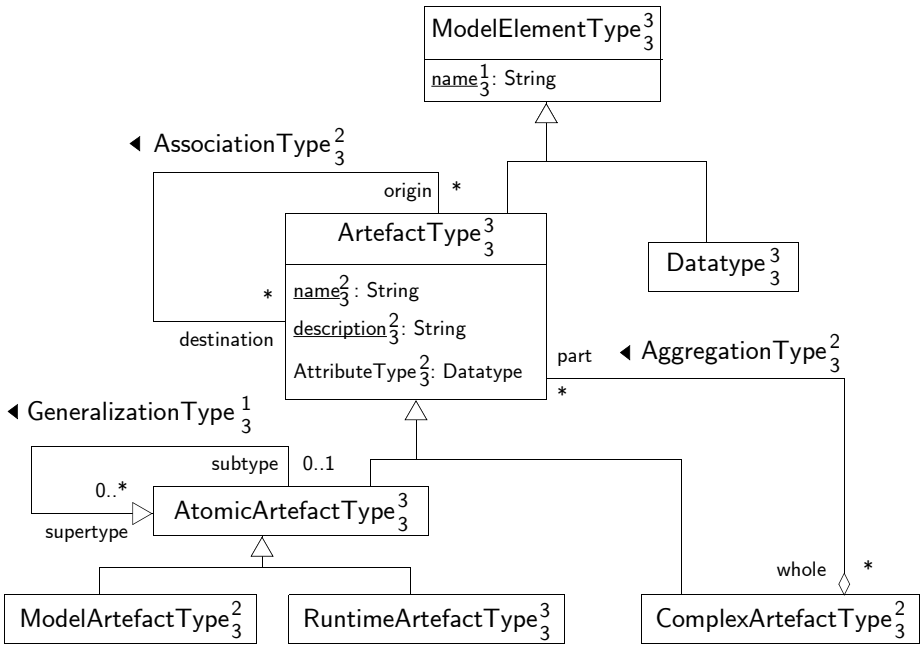
An allen Stellen, wo ein `<bezeichner>` erwartet wird, kann dieser auch wie folgt zur Laufzeit aus dem String `<bezeichnerName>` erzeugt werden:

```
#<bezeichnerName>
```

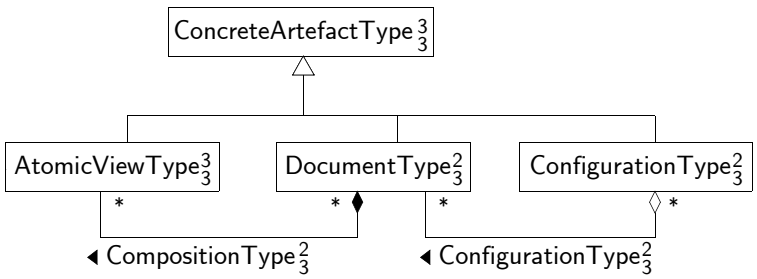
Anhang B: Produktmodelle

B.1 Produktmetamodell

B.1.1 Klassifikation der abstrakten Artefakte

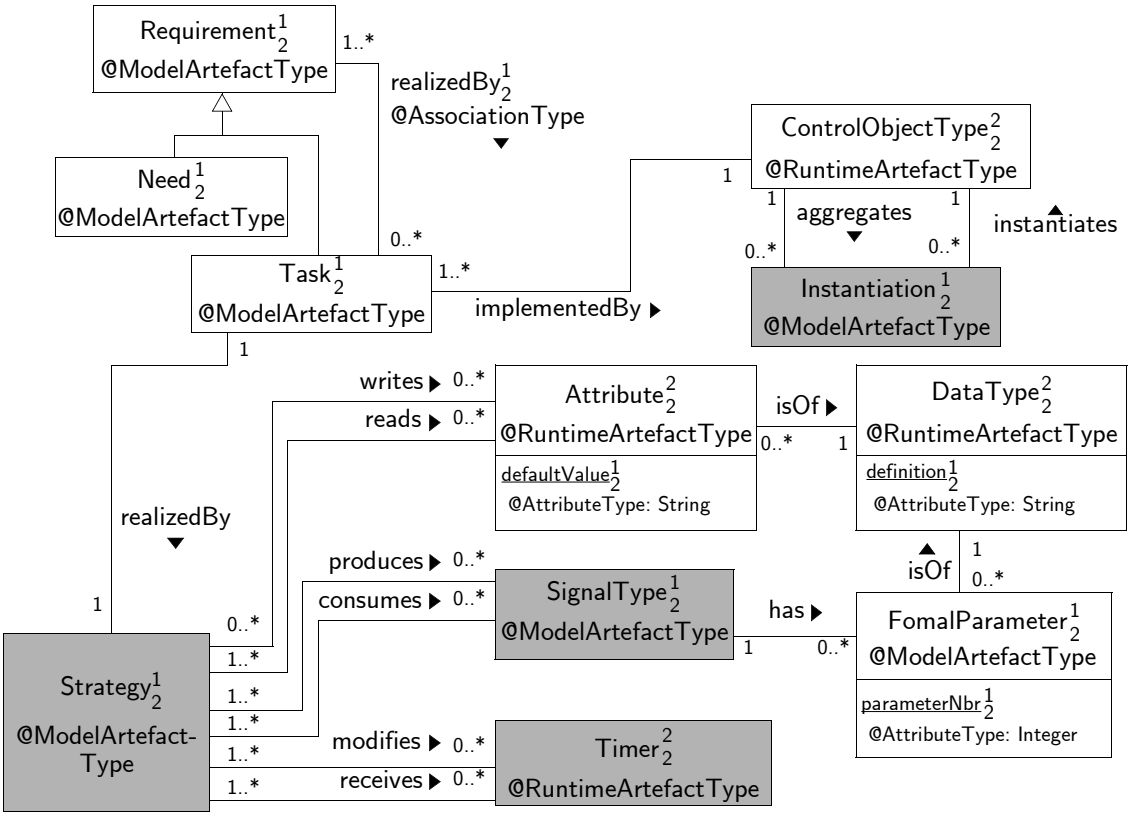


B.1.2 Klassifikation der konkreten Artefakte



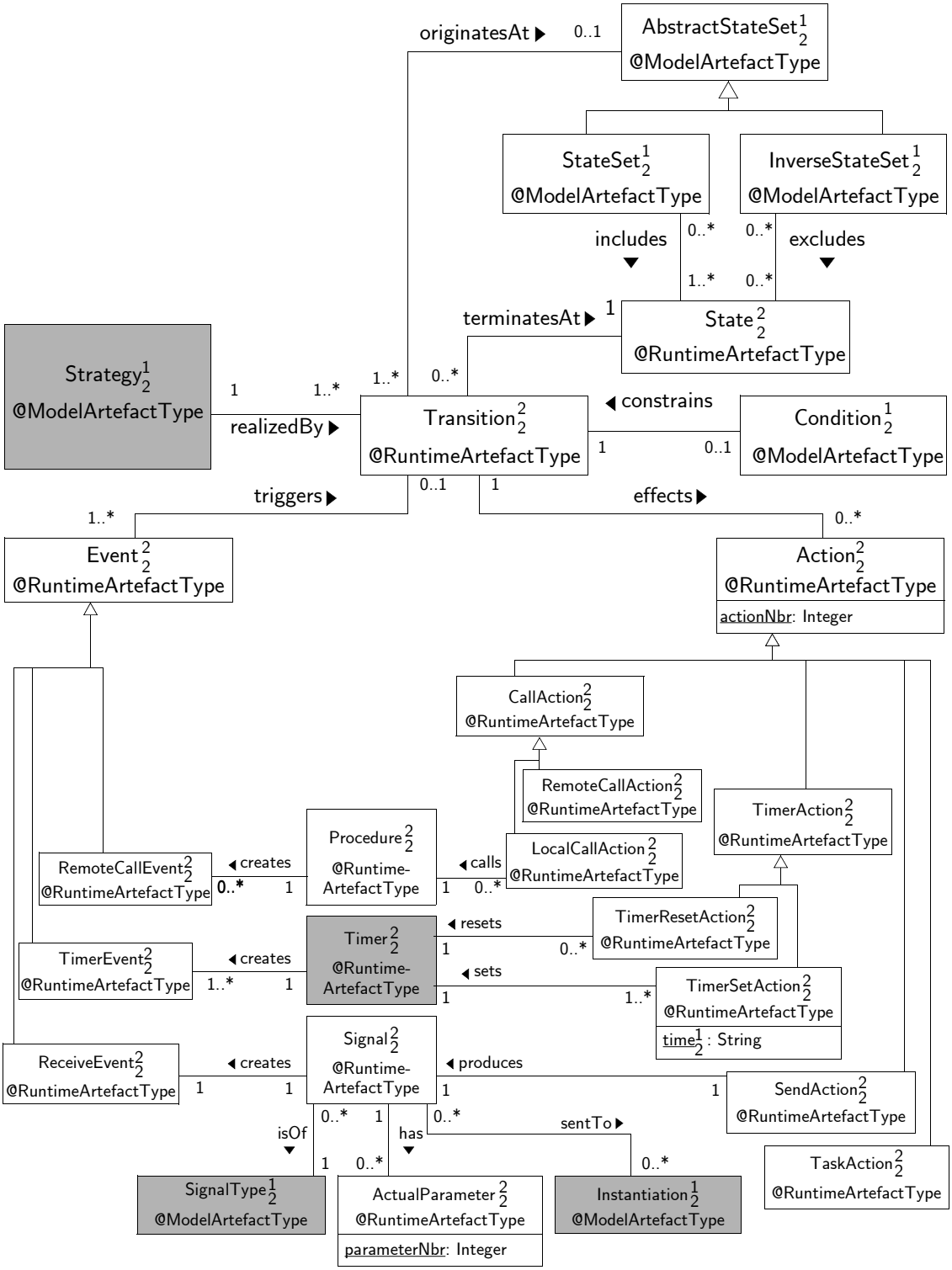
B.2 Produktmodell der PROBAnD-Methode

B.2.3 Statische Artefakttypen



Hinweis: Objekttypen, die sowohl im obigen Diagramm als auch in Diagramm B.2.4 auftauchen, sind grau hinterlegt.

B.2.4 Artefakttypen zur Verhaltensbechreibung



Literaturverzeichnis

[ACG00] Amyot, Daniel; Charfi, L.; Gorse, Nicolas et al.: "Feature Description and Feature Interaction Analysis with Use Case Maps and LOTOS" Aus: Magill, Ewan; Calder, Muffy (Hrsg.): *Feature Interactions in Telecommunications and Software Systems VI*. Proceedings Sixth International Workshop on Feature Interactions in Telecommunication and Software Systems. Glasgow, Schottland. Mai, 2000. Amsterdam (IOS Press) 2000.

[ACR03] Appukuttan, Biju; Clark, Tony; Reddy, Sreedhar et al.: "A Model Driven Approach to Model Transformations" Aus: Rensink, Arend (Hrsg.): *Workshop on Model Driven Architecture: Foundations and Applications*. Proceedings. Enschede, Niederlande. Juni, 2003. Enschede, Niederlande (Universität Twente) 2003.

[AEH99] Andries, Marc; Engels, Gregor; Habel, Annegret et al.: "Graph Transformation for Specification and Programming" In: *Science of Computer Programming*, 34. Jg. (1999), H. 1, S. 1–54.

[AKH00] Atkinson, Colin; Kühne, Thomas; Henderson-Sellers, Brian: "To Meta or Not to Meta—That Is The Question" In: *Journal of Object-Oriented Programming* (JOOP), 13. Jg. (2000),

[AKH03] Atkinson, Colin; Kühne, Thomas; Henderson-Sellers, Brian: "Systematic Stereotype Usage" In: *Software and Systems Modeling*, 2. Jg. (2003), H. 3, S. 153–163.

[AkK02] Akehurst, David; Kent, Stuart: "A Relational Approach to Defining Transformations in a Metamodel" Aus: Jezequel, Jean-Marc; Hußmann, Heinrich; Cook, Stephen (Hrsg.): *UML 2002 - The Unified Modeling Language: Model Engineering, Concepts and Tools*. Proceedings 5th International Conference. Dresden, Deutschland. September, 2002. LNCS 2460. Heidelberg (Springer-Verlag) 2002. S. 243–258.

[AlP03] Alanen, Marcus; Porres, Ivan: "Difference and Union of Models" Aus: Stevens, Perdita (Hrsg.): *UML 2003 - The Unified Modeling Language: Modeling Languages and Applications*. Proceedings 6th International Conference. San Francisco, Calif., USA. October, 2003. LNCS 2863. Berlin; Heidelberg (Springer-Verlag) 2003. S. 2–17.

[AmE03] Amyot, Daniel; Eberlein, Armin: "An Evaluation of Scenario Notations and Construction Approaches for Telecommunication Systems Development" In: *Telecommunication Systems*, 24. Jg. (2003), H. 1, S. 61–94.

[AmL03] Amyot, Daniel; Logrippo, Luigi (Hrsg.): *Feature Interactions in Telecommunications and Software Systems VII*. Proceedings Seventh International Workshop on Feature Interactions in Telecommunication and Software Systems, Ottawa, Kanada. Juni, 2003. Amsterdam (IOS Press) 2003.

[ARS97] Altmeyer, Joachim; Riegel, Jan Peter; Schürmann, Bernd et al.: "Application of a Generator-Based Software Development Method Supporting Model Reuse" Aus: Olive, Antoni; Pastor, Joan A. (Hrsg.): *Advanced Information Systems Engineering*. Proceedings 9th International Conference CAiSE'97. Barcelona, Spanien. Juni, 1997. LNCS 1250 Heidelberg (Springer-Verlag) 1997.

[ASH93] —: *1993 ASHRAE Handbook: Fundamentals*. I-P Edition. Atlanta, Ga., USA (American Society of Heating, Refregerating and Air-Conditioning Engineers, ASHRAE) 1993.

[ASU97] Aho, Alfred V.; Sethi, Ravi; Ullman, Jeffrey D.: *Compilerbau*. Teil 1. 5. unveränderter Nachdruck. Bonn; Paris; Reading, Mass. (Addison-Wesley Deutschland) 1997.

[AtK00] Atkinson, Colin; Kühne, Thomas: "Meta-level Independent Modelling" Aus: Bezivin, Jean; Ernst, Johannes (Hrsg.): *ECOOP 2000 International Workshop on Model Engineering* (IWME 2000). Proceedings. Cannes, Frankreich, Juni 2000. 2000.

[AtK01] Atkinson, Colin; Kühne, Thomas: "The Essence of Multilevel Metamodeling" Aus: Gogolla, Martin; Kobryn, Chris (Hrsg.): *UML 2001 - The Unified Modeling Language: Modeling Languages, Concepts, and Tools*. Proceedings 4th International Conference. Toronto, Kanada. Oktober, 2001. LNCS 2185. Heidelberg (Springer-Verlag) 2001. S. 19–33.

[Atk97] Atkinson, Colin: "Meta-Modeling for Distributed Object Environments" Aus: Milosevic, Zoran (Hrsg.): *1st International Enterprise Distributed Object Computing Conference* (EDOC '97). Proceedings. Gold Coast, Australien. Oktober, 1997. (IEEE Computer Society) 1997. S. 90–103.

[Atk99] Atkinson, Colin: "Supporting and Applying the UML Conceptual Framework" Aus: Bezivin, Jean; Muller, Pierre-Alain (Hrsg.): *UML '98 - The Unified Modeling Language: Beyond the Notation*. Proceedings First International Workshop. Mulhouse, Frankreich. Juni, 1998. LNCS 1618. Heidelberg (Springer-Verlag) 1999. S. 21–36.

[Aus04] Austin, Calvin: J2SE 1.5 in a Nutshell. Sun Technical Article. February 2004. `http://java.sun.com/developer/technicalArticles/releases/j2se15/`

[Aus99] Austern, Matthew H.: *Generic programming and the STL. Using and extending the C++ Standard Template Library*. Reading, Mass. u.a. (Addison-Wesley) 1999.

[BaB01] Bamberg, Günter; Baur, Franz: *Statistik*. 11., überarbeitete Auflage. München, Wien (R. Oldenbourg Verlag) 2001.

[Bak80] Bakker, Jaco de: *Mathematical Theory of Program Correctness*. Englewood Cliffs, NJ, USA (Prentice/Hall International) 1980.

[BBJ03] Baker, Paul; Bristow, Paul; Jervis, Clive et al.: "Automatic Generation of Conformance Tests from Message Sequence Charts" Aus: Sherratt, Edel (Hrsg.): *Telecommunications and beyond: The Broader Applicability of SDL and MSC*. Proceedings 3rd International Workshop on SDL and MSC (SAM). Aberystwyth, Wales. Juni , 2002. LNCS 2599. Heidelberg (Springer-Verlag) 2003. S. 170–198.

[BCG01] Boehme, R.; Covell, D.; Grant, S. et al.: "Automated Control Systems" In: *Green Hotelier Magazine*, Jg. 2001,

[BDC00] Bauer, Horst; Dietsche, Karl-Heinz; Crepin, Jürgen et al.: *BOSCH Automotive Handbook*. 5th revised and extended edition. Stuttgart (Robert Bosch GmbH) 2000.

[Ber90] Berliner, Brian: "CVS II: Parallelizing Software Development" Aus: *Proceedings of the USENIX Winter 1990 Technical Conference*. 1990.

[BFJ03] Bezivin, Jean; Farcet, Nicolas; Jezequel, Jean-Marc et al.: "Reflective Model Driven Engineering" Aus: Stevens, Perdita (Hrsg.): *UML 2003 - The Unified Modeling Language: Modeling Languages and Applications*. Proceedings 6th International Conference. San Francisco, Calif., USA. October, 2003. LNCS 2863. Berlin; Heidelberg (Springer-Verlag) 2003. S. 175–189.

[BGH99] Braek, Rolv; Gorman, Joe; Haugen, Oystein et al.: *TIMe Electronic Textbook*. Version 4.0. 1999-09-03. Trondheim, Norwegen 1999.

[BGP03] Boroday, Sergiy; Groz, Roland; Petrenko, Alex et al.: "Techniques for Abstracting SDL Specifications" Aus: Sherratt, Edel (Hrsg.): *Telecommunications and beyond: The Broader Applicability of SDL and MSC*. Proceedings 3rd International Workshop on SDL and MSC (SAM). Aberystwyth, Wales. Juni , 2002. LNCS 2599. Heidelberg (Springer-Verlag) 2003. S. 141–157.

[BHH78] Box, George, E.P.; Hunter, William G.; Hunter, J. Stuart: *Statistics for Experimenters. An Introduction to Design, Data Analysis and Model Building*. New York, Chichester, Brisbane (John Wiley & Sons) 1987.

[BHK04] Born, Marc; Holz, Eckhardt; Kath, Olaf: *Softwareentwicklung mit UML-2. Die neuen Entwurfstechniken UML 2, MOF und MDA*. München, Boston (Addison-Wesley) 2004.

[BiP92] Bischofberger, Walter R.; Pomberger, Gustav: *Prototyping-Oriented Software Development*. Concepts and Tools. Berlin, Heidelberg u.a. (Springer-Verlag) 1992.

[BKK92] Budde, Reinhard; Kautz, Karlheinz; Kuhlenkamp, Karin et al.: *Prototyping. An Aproach to Evolutionary System Development*. Berlin, Heidelberg u.a. (Springer-Verlag) 1992.

[BKP04] Böckle, Günter; Knauber, Peter; Pohl, Klaus et al.: *Software-Produktlinien – Methoden, Einführung und Praxis*. Heidelberg (dPunkt Verlag) 2004.

[Bla94] Blaschek, Günther: *Object-Oriented Programming with Prototypes*. Berlin, Heidelberg, New York (Springer-Verlag) 1994.

[BoB86] Bolognesi, Tommaso; Brinksma, Ed: "Introduction to the ISO Specification Language LOTOS" In: *Computer Networks and ISDN Systems*, Jg. 1986, H. 14, S. 295–312.

[Boc04] Bock, Conrad: "UML 2: Aktivitäten und Aktionen" In: *OBJEKTspektrum*, Jg. 2004, H. 4, S. 46–50.

[Boh04] Bohlen, Matthias: AndroMDA from UML to deployable components. Web-Site. 2004. `http://www.andromda.org/`

[Boo91] Booch, Grady: *Object Oriented Design. With Applications*. Redwood City, Calif. (Bejamin/Cummings Publishing Company, Inc.) 1991.

[BrA03] Bruda, Sefan D.; Akl, Selim G.: "Real-Time Computation: A Formal Definition and its Applications" In: *International Journal of Computers and Applications*, 25. Jg. (2003), H. 2, S. 1–11.

[Bra83] Brachman, Ronald: "What IS-A Is and Isn't: An Analysis of Taxonomic Links in Semantic Networks" In: *IEEE Computer*, 16. Jg. (1982), H. 10, S. 30–36.

[Bre01] Bredereke, Jan: "Ein Werkzeug zum Generieren von Spezifikationen aus einer Familie formaler Anforderungen" Aus: Fischer, S.; Jung, H. W. (Hrsg.): *Formale Beschreibungstechniken*. 11. GI/ITG-Fachgespräch. Bruchsal. Juni, 2001. 2001.

[BrH93] Braek, Rolv; Haugen, Oystein: *Engineering Real-Time Systems. An Object-oriented Methodology Using SDL*. New York, London u.a. (Prentice-Hall) 1993.

[BrS03] Brandt, Jens; Schäfer, Bernd Helge: "Komposition von Systemkomponenten im Rahmen des PROBAnD Framework" Aus: Metzger, Andreas; Zimmermann, Gerhard (Hrsg.): *Modellierung Reaktiver Systeme: Ein Fallbeispiel*. SFB 501 Bericht 08/03. Sonderforschungsbereich 501. Technische Universität Kaiserslautern. Kaiserslautern 2003. S. 75–85.

[BTM99] Bardohl, Roswitha; Taentzer, Gabriele; Minas, Mark et al.: "Application of Graph Transformation to Visual Languages" Aus: Rozenberg, Grzegorz (Hrsg.): *Handbook on Graph Grammars: Applications*. Vol. 2. Singapur (World Scientific) 1999. S. 105–180.

[Buh98] Buhr, R.J.A.: "Use Case Maps as Architectural Entities for Complex Systems" In: *IEEE Transactions on Software Engineering*, 24. Jg. (1998), H. 12, S. 1131–1155.

[CAD01] Carter, Ryan A.; Anton, Annie I.; Dagnino, Aldo: "Evolving Beyond Requirements Creep: A Risk-Based Evolutionary Prototyping Model" Aus: — (Hrsg.): *IEEE 5th International Symposium on Requirements Engineering* (RE '01). Proceedings. Toronto, Kanada. August, 2001. (IEEE Computer Society Press) 2001. S. 94–101.

[Cal03] Calderon Meza, Guillermo: "Modeling Friction in a Ground Vehicle - Brake Discs & Tire/Road Friction Models" Aus: Metzger, Andreas; Zimmermann, Gerhard (Hrsg.): *Modellierung Reaktiver Systeme: Ein Fallbeispiel*. SFB 501 Bericht 08/03. Sonderforschungsbereich 501. Technische Universität Kaiserslautern. Kaiserslautern 2003. S. 37–50.

[CaS02] Caplat, Guy; Sourrouille, Jean Louis: "Model Mapping in MDA" Aus: Bezivin, Jean; France, Robert (Hrsg.): *WiSME 2002*. Proceedings UML 2002 Workshop in Software Model Engineering. Dresden, Deutschland. Oktober, 2002. u. J.

[CaS03] Caplat, Guy; Sourouille, Jean Louis: "Considerations about Model Mapping" Aus: Bezivin, Jean; Gogolla, Martin (Hrsg.): *WiSME 2003*. Proceedings UML 2003 Workshop in Software Model Engineering. San Francisco, Calif., USA. October, 2003. 2003.

[Cer02] Cerami, Ethan: *Web Services Essentials*. Beijing; Cambridge; Farnham et al. (O'Reilly) 2002.

[CKM03] Calder, Muffy; Kolberg, Mario; Magill, Evan: "Feature Interaction: A Critical Review and Considered Forecast" In: *Computer Networks*, 41. Jg. (2003), H. 1, S. 115–141.

[ClE97] Clark, Tony; Evans, Andy: "Foundations of the Unified Modeling Language" Aus: Duke, David; Evans, Andy (Hrsg.): *Bcs-Facs Northern Formal Methods Workshop*. Proceedings. Ilkley, Großbritannien, September, 1996. Heidelberg (Springer-Verlag) 1997.

[CoC01] Cousot, Patrick; Cousot, Radhia: "Verification of Embedded Software: Problems and Perspectives" Aus: Henzinger, T.A.; Kirsch C.M. (Hrsg.): *EMSOFT 2001*. Proceedings Embedded Software. Tahoe City, Kalifornien, USA. Oktober, 2001. LNCS 2211. Heidelberg (Springer-Verlag) 2001. S. 97–113.

[Coc02] Cockburn, Alistair: *Agile Software Development*. Boston (Addison-Wesley) 2002.

[CoD94] Cook, Steve; Daniels, John: *Designing Object Systems. Object-Oriented Modeling with Syntropy*. New York, London, Toronto (Prentice Hall) 1994.

[Com04] —: "Code-Generierung zwingt zum Umdenken" In: *Computer Zeitung*, Nr. 33/34 vom 16. August 2004.

[CST00] Cazzola, Walter; Sosio, Andreas; Tisato, Francesco: "Shifting UP Reflection from the Implementation to the Analysis Level" Aus: Cazzola, Walter et al. (Hrsg.): *Reflection and Software Engineering*. LNCS 1826. Heidelberg (Springer-Verlag) 2000. S. 1–20.

[Cza03] Czarnecki, Krzysztof: "Perspectives on Generative Programming" Aus: Alt, Helena; Becker, Martin (Hrsg.): *Software Reuse: Requirements, Technologies and Applications*. Proceedings. International Colloquium of the SFB 501. Kaiserslautern. März, 2003. Kaiserslautern (Universität Kaiserslautern) 2003. S. 81–88.

[CzE00] Czarnecki, Krzysztof; Eisenecker, Ulrich W.: *Generative Programming: Methods, Tools and Applications*. Boston, San Francisco, New York (Addison-Wesley) 2000.

[CzH03] Czarnecki, Krzysztof; Helsen, Simon: "Classification of Model Transformation Approaches" Aus: Bettin, Jorn; van Emde Boas, Ghica; Agrawal, Aditya et al. (Hrsg.): *OOPSLA 2003 2nd Workshop on Generative Techniques in the Context of MDA*. Proceedings. Anaheim, Kalif., USA. October, 2003. 2003.

[DaC03] Davies, Jim; Crichton, Charles: "Using State Diagrams to Describe Concurrent Behavior" Aus: Dong, J.S.; Woodcock, J. (Hrsg.): *ICFEM 2003*. LNCS 2885. Heidelberg (Springer-Verlag) 2003. S. 105–124.

[DeS00] Dehnert, James C.; Stepanov, Alexander: "Fundamentals of Generic Programming" Aus: Jazayeri, Mehdi; Loos, Rüdiger G.K.; Musser, David R. (Hrsg.): *Generic Programming*. Proceedings. International Seminar on Generic Programming. Schloss Dagstuhl, Deutschland. April und Mai, 1998. LNCS 1766. Heidelberg (Springer-Verlag) 2000. S. 1–11.

[Dew94] Dew, Robert A.: “A Rapid Prototyping System for Real-Time Systems” Aus: — (Hrsg.): *ACM 22rd Annual Computer Science Conference on Scaling up: Meeting the Challenge of Complexity in Real-World Computing Applications*. Proceedings CSC ’94. Phoenix, Arizona, USA. März, 1994. (ACM) 1994. S. 7–14.

[DIN95] —: *Begriffe zu Qualitätsmanagement und Statistik - Teil 11: Begriffe des Qualitätsmanagements*. DIN 55350-11, Ausgabe:1995-08. Berlin (DIN Deutsches Institut für Normung e. V.) 1995.

[Dir02] Dirckze, Ravi et al.: *Java Metadata Interface(JMI) Specification*. JSR 040. Java Community Process. Version 1.0. Final Specification. Juni, 2002. 2002.

[DLR03] Dominguez, Eladio; Lloret, Jorge; Rubio Angel L. et al.: “An MDA-Based Approach to Managing Database Evolution” Aus: Rensink, Arend (Hrsg.): *Workshop on Model Driven Architecture: Foundations and Applications*. Proceedings. Enschede, Niederlande. Juni, 2003. Enschede, Niederlande (Universität Twente) 2003.

[Dol03] Doldi, Laurent: *Validation of Communications Systems with SDL. The Art of SDL Simulation and Reachability Analysis*. Chichester, West Sussex, England (John Wiley & Sons Ltd.) 2003.

[Dou00] Douglass, Bruce P.: *Real-Time UML - Second Edition. Developing Efficient Objects for Embedded Systems*. Reading, Mass., USA (Addison-Wesley) 2000.

[Dou99] Douglass, Bruce P.: *Doing Hard Time. Developing Real-Time Systems with UML, Objects, Frameworks, and Patterns*. Reading, Mass., USA (Addison-Wesley-Longman) 1999.

[Dre93] Drees, Horst: *UNIX: Das Kompendium. Einführung, Arbeitsbuch, Nachschlagewerk*. Haar bei München (Markt&Technik Buch- und Software-Verlag) 1993.

[Ebe98] Ebert, Christof: “Experiences with Colored Predicate-Transition Nets for Specifying and Prototyping Embedded Systems” In: *IEEE Transactions on Systems, Man and Cybernetics*, Part B (Cybernetics), 28. Jg. (1998), H. 5, S. 641–652.

[Eck02] Eckel, Bruce: *Thinking in Java*. 3rd Edition. Upper Saddle River, NJ (Prentice-Hall) 2002.

[EgG02] Egyed, Alexander; Grünbacher, Paul: “Automating Requirements Traceability: Beyond the Record & Replay Paradigm” Aus: — (Hrsg.): *ASE 2002*. Proceedings 17th IEEE International Conference on Automated Software Engineering. Edinburgh, Scotland, UK. September, 2002 (IEEE) 2002. S. 163–171.

[EHS97] Ellsberger, Jan; Hogrefe, Dieter; Sarma, Amardeo: *SDL – Formal Object-oriented Language for Communicating Systems*. London, New York, u.a. (Prentice Hall) 1997.

[EIA04] EIA/CDIF: *CDIF – Understanding between modelling tools*. Web-Site. 2004. `http://www.eigroup.org/cdif/index.html`

[Eic91] Eicker, Stefan: “Anforderungen an ein zentrales Metadatenbanksystem” In: *HMD Theorie und Praxis der Wirtschaftsinformatik*, 28. Jg. (1991), H. 161, S. 3–9.

[EJT04] Edwards, Jonathan; Jackson, Daniel; Torlak, Emina: “A Type System for Object Models” erscheint in *ACM SIGSOFT Conference on Foundations of Software Engineering*. Newport Beach, Calif., USA. November, 2004.

[EnK00] Engstrom, Eric; Krueger, Jonathan: “Building and Rapidly Evolving Domain-Specific Tools with DOME” Aus: — (Hrsg.): *IEEE International Symposium on Computer-Aided Control Systems Design*. Proceedings. Anchorage, Alaska. Septermber, 2000. (IEEE Computer Society Press) 2000. S. 65–70.

[Fen91] Fenton, Norman E.: *Software Metrics. A Rigorous Approach*. London, New York u.a. (Chapman & Hall) 1991.

[FGH94] Finkelstein, Anthony; Gabbay, Dov; Hunter, Anthony et al.: “Inconsistency Handling in Multi-Perspective Specifications” In: *IEEE Transactions on Software Engineering*, 20. Jg. (1994), H. 8, S. 569–578.

[FGM01] Franconi, Enrico; Grandi, Fabio; Mandreoli, Federica: "Schema Evolution and Versioning: A Logical and Computational Characterisation" Aus: Balsters, Herman; De Brock, Bert; Conrad, Stefan (Hrsg.): *Database Schema Evolution and Meta-Modeling*. Proceedings 9th International Workshop on Foundations of Models and Languages for Data and Objects (FoMLaDO/DEMM 2000). Dagstuhl. September, 2000. Heidelberg (Springer-Verlag) 2001. S. 85–99.

[FGS03] France, Robert; Ghosh, Sudipto; Song, Eunjee et al.: "A Metamodeling Approach to Pattern-Based Model Refactoring" In: *IEEE Software*, September/Oktober. Jg. (2003), S. 52–58.

[Fla02] Flatscher, Rony G.: "Metamodeling in EIA/CDIF—Meta-Metamodel and metamodels" In: *ACM Transactions on Modeling and Computer Simulation* (TOMACS), 12. Jg. (2002), H. 4, S. 322–342.

[Flo84] Floyd, Christiane: "A Systematic Look at Prototyping" Aus: Budde, Reinhard (Hrsg.): *Approaches To Prototyping*. Proceedings Working Conference on Prototyping. Namur, Belgien. Oktober, 1984. Berlin, Heidelberg (Springer-Verlag) 1984. S. 1–18.

[Fra99] Frankel, David S.: "Business Objects: The OMG Meta Object Facility" In: *Java Report*, Jg. 1999,

[FrR04] France, Robert; Rumpe, Bernhard: "Editorial: In Search of Effective Design Abstractions" In: *Software and Systems Modeling*, 3. Jg. (2004), H. 1, S. 1–3.

[Gab98] Gabler: *Wirtschaftslexion*. CD-ROM Version. Wiesbaden (Gabler) 1998.

[GaN00] Gansner, Emden R.; North, Stephen C.: "An Open Graph Visualization System and its Applications to Software Engineering" In: *Software Practice and Experience*, 30. Jg. (2000), H. 11, S. 1203–1233.

[Gen99] Genilloud, Guy: "On the Abstraction of Objects, Components and Interfaces" In: — (Hrsg.) *OOPSLA Workshop on Behavior Semantics*. Proceedings. Denver, Colorado, USA. November, 1999. 1999.

[GeW00] Genilloud, Guy; Wegmann, Alain: "On Types, Instances, and Classes in UML" Aus: Moreira, A.; Bruel, J.M.; France, R. (Hrsg.): *ECOOP Workshop on Defining Precise Semantics for UML*. Proceedings 14th European Conference on Object-Oriented Programming. Sophia-Antipolis, Cannes, Frankreich. Juni, 2000. 2000.

[GGK03] Gardner, Tracy; Griffin, Catherine; Koehler, Jana et al.: "A review of OMG MOF 2.0 Query/Views/Transformations Submissions and Recommendations towards the final Standard" Aus: Evans, Andy; Sammut, Paul; Willans, James S. (Hrsg.): *Metamodelling for MDA*. Proceedings. First International Workshop. York, UK. November, 2003. 2003. S. 178–197.

[GHJ97] Gamma, Erich; Helm, Richard; Johnson, Ralph E.: *Design Patterns. Elements of Reusable Object-Oriented Software*. Reading, Mass., USA (Addison-Wesley) 1997.

[GHP02] Gery, Eran; Harel, David; Palachi, Eldad: "Rhapsody: A Complete Life-Cycle Model-Based Development System" Aus: Butler, M.; Petre, L.; Sere, K. (Hrsg.): *Integrated Formal Methods: Third International Conference*. Proceedings IFM 2002. Turku, Finnland. Mai, 2002. LNCS 2335. Heidelberg (Springer-Verlag) 2002. S. 1–10.

[GHR03] Goldmann, Siegrid; Holz, Harald; Richter Michael M.: "Knowlegde Management for Project Planning and Enactment in Software Engineering" Aus: Alt, Helena; Becker, Martin (Hrsg.): *Software Reuse: Requirements, Technologies and Applications*. Proceedings. International Colloquium of the SFB 501. Kaiserslautern. März, 2003. Kaiserslautern (Universität Kaiserslautern) 2003. S. 59–70.

[GJM91] Ghezzi, Carlo; Jazayeri, Mehdi; Mandrioli, Dino: *Fundamentals of Software Engineering*. Englewood Cliffs, NJ, USA (Prentice Hall) 1991.

[GLR02] Gerber, Anna; Lawley, Michael; Raymond, Kerry et al.: "Transformation: The Missing Link of MDA" Aus: Corradini, Andrea; Ehrig, Hartmut; Kreowski, Hans-Jörg et al. (Hrsg.): *Graph Transformation: First International Conference*. Proceedings ICGT 2002. Barcelona, Spanien. Oktober, 2002. LNCS 2505. Heidelberg (Springer-Verlag) 2002. S. 90–105.

[GoF94] Gotel, Orlena C.Z.; Finkelstein, Anthony: "An Analysis of the Requirements Traceability Problem" Aus: — (Hrsg.): *International Conference on Requirements Engineering*. Proceedings. April, 1994. Colorado Springs, Colorado, USA. LosAlamitos, Calif. (IEEE Computer Society Press) 1994. S. 94–101.

[GoR99] Gogolla, Martin; Richters, Mark: "Equivalence Rules for UML Class Diagrams" Aus: Bezivin, Jean; Muller, Pierre-Alain (Hrsg.): *UML '98 - The Unified Modeling Language: Beyond the Notation*. Proceedings First International Workshop. Mulhouse, Frankreich. Juni, 1998. LNCS 1618. Heidelberg (Springer-Verlag) 1999. S. 87–96.

[Gra03] Graf, Susanne: "Expression of Time and Duration Constraints in SDL" Aus: Sherratt, Edel (Hrsg.): *Telecommunications and beyond: The Broader Applicability of SDL and MSC*. Proceedings 3rd International Workshop on SDL and MSC (SAM). Aberystwyth, Wales. Juni , 2002. LNCS 2599. Heidelberg (Springer-Verlag) 2003. S. 38–52.

[Gra92] Grady, Robert B.: *Practical Software Metrics for Project Management and Process Improvement*. Englewood Cliff, NJ, USA (Prentice Hall, Inc.) 1992.

[Han03] Hanke, Wolfgang: *Modellbasierte Konsistenzprüfung und -herstellung von Entwicklungsdokumenten*. Diplomarbeit. Fachbereich Informatik. Technische Universität Kaiserslautern. (TU Kaiserslautern) 2003.

[HaR00] Harel, David; Rumpe, Bernhard: *Modeling Languages: Syntax, Semantics and All That Stuff - Part I: The Basic Stuff*. Report MCS00-16. Rehovot, Israel 2000.

[Har01] Harel, David: "From Play-In Scenarios to Code: An Achievable Dream" In: *IEEE Computer*, Jg. 2001, H. Januar, S. 1–8.

[Har03] Harper, Richard (Hrsg.): *Inside the Smart Home*. Heidelberg (Springer-Verlag) 2003.

[HeB99] Henderson-Sellers, Brian; Barbier, Franck: "Black and White Diamonds" Aus: France, Robert; Rumpe, Bernhard (Hrsg.): *UML '99 - The Unified Modeling Language: Beyond the Standard*. Proceedings. 2nd International Conference. Fort Collins, Colorado, USA. Oktober, 1999. LNCS 1723. Heidelberg (Springer-Verlag) 1999. S. 550–565.

[HeH02] Helsel, D. R.; Hirsch, R. M.: *Statistical Methods in Water Resources*. Chapter A3. Book4. Techniques of Water-Resources Investigations of the United States Geological Survey. (USGS, U.S. Department of the Interior) 2002.

[Hei04] Heitmeyer, Constance: "Managing Complexity in Software Development with Formally Based Tools" Aus: *Proceedings FESCA 2004*. ETAPS Satellite Event. Electronic Notes in Computer Science. (Elsevier) 2004.

[HeI88] Hekmatpour, Sharam; Ince, Darrel C.: *Software prototyping, formal methods and VDM*. Workingham (Addison-Wesley) 1988.

[Hen04] Henkel, Hans-Olaf: *Die Ethik des Erfolgs. Spielregeln für die globalisierte Gesellschaft*. München (Ullstein Verlag) 2004.

[HHS02] Hausmann, Jan Hendrik; Heckel, Reiko; Sauer, Stefan: "Extended Model Relations with Graphical Consistency Conditions" Aus: Kuzniarz, Ludwik; Reggio, Gianna; Sourrouille, Jean Louis et al. (Hrsg.): *UML 2002 Workshop on Consistency Problems in UML-based Software Development*. Proceedings. Dresden, Deutschland. September/Oktober, 2002. 2002. S. 61–73.

[HJG99] Ho, Wai Ming; Jezequel, Jean-Marc; Le Guennec, Alain et al.: "UMLAUT: an Extendible UML Transformation Framework" Aus: Setliff, Dorothy; Hall, Robert J.; Tyugu, Enn (Hrsg.): *14th IEEE International Conference on Automated Software Engineering*. Proceedings. Cocoa Beach, Florida, USA. Oktober, 1999. 1999. S. 275–278.

[Hof98] Hoffmann, Josef: *MATLAB und Simulink. Beispielorientierte Einführung in die Simulation dynamischer Systeme*. Bonn u.a. (Addison-Wesley-Longman) 1998.

[Hol03] Holz, Eckhardt: "Strategien zur Unterstützung der Modellkombination im Softwareentwicklungsprozess" Aus: Metzger, Andreas; Zimmermann, Gerhard (Hrsg.): *Modellierung Reaktiver Systeme: Ein Fallbeispiel*. SFB 501 Bericht 08/03. Sonderforschungsbereich 501. Technische Universität Kaiserslautern. Kaiserslautern 2003. S. 3–18.

[HoU88] Hopcroft, John E.; Ullman, Jeffrey D.: *Einführung in die Automatentheorie, Formale Sprachen und Komplexitätstheorie*. Bonn; Reading, Mass.; Menlo Park, Calif. (Addison-Wesley Publishing Company) 1988.

[HrR02] Hruschka, Peter; Rupp, Chris: *Agile Softwareentwicklung für Embedded Real-Time Systems mit der UML*. München, Wien (Carl Hanser Verlag) 2002.

[Hub02] Hubert, Richard: *Convergent Architecture. Building Model-Driven J2EE Systems with UML*. New York, Chichester u.a. (Wiley Computer Publishing) 2002.

[Ill04] Illgen, Thomas: "Entwicklung eingebetteter Systeme in der Automotive-Industrie" Vortrag. LPZ Distinguished Lecture Series. 14.04.2004. Lehrstuhl für Angewandte Telematik/E-Business. Leipzig 2004.

[ISO01] —: *Software Engineering – Product Quality – Part 1: Quality model*. ISO/IEC 9126-1:2001 Genf (International Organization for Standardization, ISO) 2001.

[ITU99] —: *Specification and description language (SDL)*. Recommendation Z.100 (11/99). Genf (International Telecommunication Union) 1999.

[IyM04] Iyengar, Sridhar; Merks, Ed: "Rapid plug-in development and integration using EMF" EclipseCON Presentation. Anaheim, Calif, USA. 2004. 2004.

[Jal97] Jalote, Pankaj: *An Integrated Approach to Software Engineering*. Second Edition. New York (Springer-Verlag) 1997.

[JRH03] Jeckle, Mario; Rupp, Chris; Hahn, Jürgen et al.: *UML 2 glasklar*. München (Hanser Fachbuchverlag) 2003.

[Kas01] Kaschek, Roland: "Modellbestandteile: Modellierungsbegriffe und Abstraktionsbegriffe" Aus: Engels, Gregor; Oberweis, Andreas; Zündorf, Albert (Hrsg.): *Modellierung 2001*. Proceedings Workshop der Gesellschaft für Informatik e.V. (GI). Bad Lippspringe, März, 2001. Lecture Notes in Informatics (LNI) P-1. Bonn (Köllen-Verlag) 2001. S. 138–147.

[KCR98] Kelsey, Richard; Clinger, William; Rees, Jonathan: "Revised Report on the Algorithmic Language Scheme" In: *ACM SIGPLAN Notices*, 33. Jg. (1998), H. 9, S. 26–76.

[KeK98] Keck, Dirk O.; Kühn, Paul J.: "The Feature and Service Interaction Problem in Telecommunications Systems: A Survey" In: *IEEE Transactions on Software Engineering*, 24. Jg. (1998), H. 10, S. 779–796.

[Ken02] Kent, Stuart: "Model Driven Engineering" Aus: Butler, M.; Petre, L.; Sere, K. (Hrsg.): *Integrated Formal Methods: Third International Conference*. Proceedings IFM 2002. Turku, Finnland. Mai, 2002. LNCS 2335. Heidelberg (Springer-Verlag) 2002. S. 286–298.

[KeT00] Kelly, Steven; Tolvanen, Juha-Pekka: "Visual Domain-Specific Modelling: Benefits and Experiences of using metaCASE Tools" Aus: Bezivin, Jean; Ernst, Johannes (Hrsg.): *ECOOP 2000 International Workshop on Model Engineering* (IWME 2000). Proceedings. Cannes, Frankreich, Juni 2000. 2000.

[KFF86] Kohlbecker, E.; Friedman, D.P.; Felleisen, M. et al.: "Hygienic Macro Expansion" Aus: — (Hrsg.): *ACM Conference on LISP and Functional Programming*. Proceedings. Cambridge, Mass., USA. New York (ACM Press) 1986. S. 151–161.

[KlW03] Kleppe, Anneke; Warmer, Jos: "Do MDA Transformations Preserve Meaning? An Investigation into Preserving Semantics" Aus: Evans, Andy; Sammut, Paul; Willans, James S. (Hrsg.): *Metamodelling for MDA*. Proceedings. First International Workshop. York, UK. November, 2003. 2003. S. 13–23.

[KoA03] Kordon, Fabrice; Athanas, Peter (Hrsg.): *14th IEEE International Workshop on Rapid System Prototyping* (RSP'03). Proceedings. San Diego, Calif., USA. Juni, 2003. (IEEE Computer Society Press) 2003.

[KoB93] Kotler, Philip; Bliemel, Friedhelm: *Marketing-Management*. Analyse, Planung, Umsetzung und Steuerung. 7., vollständig neu bearbeitete und für den deutschen Sprachraum erweiterte Auflage. 5., verbesserter Nachdruck. Stuttgart (Schäffer-Poeschel Verlag) 1993.

[Kor94] Kordon, Fabrice: "Proposal for a Generic Prototyping Approach" Aus: Fujita, Hiroyuki (Hrsg.): *IEEE Symposium on Emerging Technologies & Factory Automation*. Novel Disciplines for the next Century. Proceedings ETFA '94. Tokyo, Japan. November, 1994. Piscataway, N.J. (IEEE) 1994. S. 396–403.

[Krä04] Krämer, Marc: *Eine Fallstudie zur Modellierung einer Gebäudesimulation*. Projektarbeit. Fachbereich Informatik. Technische Universität Kaiserslautern. Kaiserslautern 2004.

[Kra97] Kranz, Hans R.: "Gebäudeautomation im Wandel" In: *atp – Automatisierungstechnische Praxis*, Jg. 1997, H. 3, S. 10–15.

[Küh03] Kühne, Thomas: "Automatisierte Softwareentwicklung mit Modellcompilern" In: *Magazin "thema Forschung" der Technischen Universität Darmstadt*, 1. Jg. (2003), S. 116–123.

[Küh03a] Kühne, Thomas: "Re: FW: Deep Instantiation" E-Mail Kommunikation. 11.11.2003. 16.49 MEZ. Kaiserslautern/Darmstadt 2003.

[KüS04] Kühne, Thomas; Steimann, Friedrich: "Tiefe Charakterisierung" Aus: Rumpe, Bernhard; Hesse, Wolfgang (Hrsg.): *Modellierung 2004*. Proceedings. Tagung Modellierung. Marburg, März, 2004. Lecture Notes in Informatics (LNI). P-45. Bonn (Köllen Druck+Verlag GmbH) 2004. S. 109–120.

[KWB03] Kleppe, A.; Warmer, J.; Bast, W.: *MDA Explained: the Practice and Promise of Model-Driven Architecture*. Boston, Mass. (Addison-Wesley) 2003.

[KWJ04] King, Ross D.; Whelan, Kenneth E.; Jones, Ffion M.: "Functional Genomic Hypothesis Generation and Experimentation by a Robot Scientist" In: *nature*, 427. Jg. (2004), H. 6971, S. 247–252.

[LaC04] Lange, Christian; Chaudron, Michel: "Konsistenz und Unvollständigkeit industrieller UML Modelle" Aus: Rumpe, Bernhard; Hesse, Wolfgang (Hrsg.): *Modellierung 2004*. Proceedings. Tagung Modellierung. Marburg, März, 2004. Lecture Notes in Informatics (LNI). P-45. Bonn (Köllen Druck+Verlag GmbH) 2004. S. 295–296.

[LaM96] Laitenberger, Oliver; Münch, Jürgen: *Ein Prozeßmodell zur experimentellen Erprobung von Software-Entwicklungsprozessen*. SFB Bericht 04/06. Sonderforschungsbereich 501. FB Informatik. Kaiserslautern (Universität Kaiserslautern) 1996.

[LCA02] Lano, Kevin; Clark, David; Androutsopoulos, Kelly: "Formalising Inter-model Consistency of the UML" Aus: Kuzniarz, Ludwik; Reggio, Gianna; Sourrouille, Jean Louis et al. (Hrsg.): *UML 2002 Workshop on Consistency Problems in UML-based Software Development*. Proceedings. Dresden, Deutschland. September/Oktober, 2002. 2002. S. 133–148.

[LCM03] Lange, Christian; Chaudron, M.R.V.; Muskens, J. et al.: "An Empirical Investigation in Quantifying Inconsistency and Incompleteness of UML Designs" Aus: Kuzniarz, Ludwik; Huzar, Zbigniew; Reggio, Gianna et al. (Hrsg.): *UML 2003 Workshop on Consistency Problems in UML-based Software Development*. Proceedings. San Francisco, Kalif., USA. Oktober, 2003. 2003. S. 26–34.

[Leb04] Leblanc, Philippe: *UML 2.0 Action Semantics and Telelogic TAU/Architect and TAU/Developer Action Language*. White Paper. Malmö, Schweden (Telelogic AB) 2004.

[LEH00] Leblanc, Philippe; Ek, Anders; Hjelm, Thomas: "Telelogic SDL and MSC Tool Families" In: *Telektronikk*, Jg. 2000, H. 4, S. 156–163.

[LEM02] Liu, WenQian; Easterbrook, Steve; Mylopoulos, John: "Rule-Based Detection of Inconsistency in UML Models" Aus: Kuzniarz, Ludwik; Reggio, Gianna; Sourrouille, Jean Louis et al. (Hrsg.): *UML 2002 Workshop on Consistency Problems in UML-based Software Development*. Proceedings. Dresden, Deutschland. September/Oktober, 2002. 2002. S. 106–121.

[Lie04] Liebermann, Sascha: "Freiheit statt Vollbeschäftigung" Online-Dokument. 2004. `http://www.freiheitstattvollbeschaeftigung.de/`

[Lig02] Liggesmeyer, Peter: *Software-Qualität. Testen, Analysieren und Verifizieren von Software*. Heidelberg, Berlin (Spektrum Akademischer Verlag) 2002.

[LJK94] Lutz, Peter; Jenisch, Richard; Klopfer, Heinz et al.: *Lehrbuch der Bauphysik. Schall, Wärme, Feuchte, Licht, Brand, Klima*. Stuttgart (B.G. Teubner) 1994.

[LMB01] Ledeczi, Akos; Maroti, Milos; Bakay, Arpad et al.: "The Generic Modeling Environment" Aus: — (Hrsg.): *IEEE International Workshop on Intelligent Signal Processing*. Proceedings WISP '01. Budapest, Ungarn. Mai, 2001. (IEEE) 2001.

[LMR98] Leue, Stefan; Merhmann, Lars; Rezai, Mohammed: *Synthesizing ROOM Models from Message Sequence Chart Specifications*. Technical Report 98-06. Electrical and Computer Engineering. Ontario, Kanada (University of Waterloo) 1998.

[Löw01] Löwy, Juval: *COM and .NET component services. Mastering COM+ services*. Beijing; Köln u.a. (O'Reilly) 2001.

[Lud03] Ludewig, Jochen: "Models in Software Engineering - An Introduction" In: *Software and Systems Modeling*, 2. Jg. (2003), H. 1, S. 5–14.

[MaC00] Magill, Ewan; Calder, Muffy (Hrsg.): Feature Interactions in Telecommunications and Software Systems VI. *Proceedings Sixth International Workshop on Feature Interactions in Telecommunication and Software Systems*. Glasgow, Schottland. Mai, 2000. Amsterdam (IOS Press) 2000.

[MaL87] Marcotty, Michael; Ledgard, Henry: *The World of Programming Languages*. New York, Heidelberg (Springer-Verlag) 1987.

[MaO92] Martin, James; Odell, James: *Object-Oriented Analysis and Design*. Englewood Cliffs, N.J., USA (Prentice Hall) 1992.

[MaP96] Maffeis, Olivier; Poigne, Axel: *Synchronous Automata for Reactive, Real-Time or Embedded Systems*. Technischer Bericht Nr. 967. Forschungszentrum Informationstechnik (GMD). Januar 1996. Sankt Augustin 1996.

[MaR97] Mauw, Sjouke; Reniers, M.A.: "High-level Message Sequence Charts" Aus: Cavalli, A.; Sarma, A. (Hrsg.): *SDL'97: Time for Testing - SDL, MSC and Trends*. Proceedings 8th SDL Forum. Evry, Frankreich. September, 1997. Amsterdam, Niederlande (Elsevier) 1997. S. 291–306.

[Mau02] Maurer, Frank: "What is the Fuzz about Agile Software Processes?" Kolloquiumsvortrag. Fachbereich Informatik. Universität Kaiserslautern. 14.11.2002. 2002.

[May90] von Mayrhauser, Anneliese: *Software Engineering. Methods and Management*. Boston, San Diego, New York (Academic Press, Inc.) 1990.

[MaZ99] Mansurov, Nikolai; Zhukov, D.: "Automatic Synthesis of SDL Models in Use Case Methodology" Aus: Dssouli, Rachida; von Bochmann, Gregor; Lahav, Yair (Hrsg.): *SDL' 99. The Next Millennium*. Proceedings 9th SDL Forum. Montreal, Quebec, Kanada. Juni, 1999. Amsterdam, Niederlande (Elsevier) 1999. S. 225–240.

[MDG04] Moore, Bill; Dean, David; Gerber, Anna et al.: Eclipse Development. *Using the Graphical Editing Framework and the Eclipse Modeling Framework*. IBM RedBooks. (IBM Corp.) 2004.

[MeQ01] Metzger, Andreas; Queins, Stefan: "A Reuse- and Prototyping-based Approach for the Specification of Building Automation Systems" Aus: Schürr, Andy (Hrsg.): *Workshop on Object-Oriented Modeling of Embedded Real-Time Systems* (OMER-2). Proceedings. Herrsching. Mai, 2001. Report Nr. 2001-03. München 2001. S. 3–9.

[MeQ02] Metzger, Andreas; Queins, Stefan: "Specifying Building Automation Systems with PROBAnD, a Method Based on Prototyping, Reuse, and Object-orientation" Aus: Hofmann, Peter P.; Schürr, Andy (Hrsg.): *OMER - Object-oriented Modeling of Embedded Systems*. Proceedings GI-Workshops OMER-1 & OMER-2. Herrsching am Ammersee. Mai 1999 und Mai 2001. Lecture Notes in Informatics (LNI) P-5. Bonn (Köllen Verlag) 2002. S. 135–140.

[MeQ02a] Metzger, Andreas; Queins, Stefan: "Early Prototyping of Reactive Systems Through the Generation of SDL Specifications from Semi-formal Development Documents" Aus: Sherratt, Edel; Reed, Rick (Hrsg.): *3rd SAM Workshop*. Proceedings. University of Wales, Aberystwyth, Wales. Juni, 2002. Aberystwyth 2002.

[MeQ03] Metzger, Andreas; Queins, Stefan: "Model-Based Generation of SDL Specifications for the Early Prototyping of Reactive Systems" Aus: Sherratt, Edel (Hrsg.): *Telecommunications and beyond: The Broader Applicability of SDL and MSC*. Proceedings 3rd International Workshop on SDL and MSC (SAM). Aberystwyth, Wales. Juni , 2002. LNCS 2599. Heidelberg (Springer-Verlag) 2003. S. 158–169.

[Met01] Metzger, Andreas: *Ein flexibles Testfeld für Experimente im Bereich der Gebäudeautomation und -simulation*. SFB 501-Bericht Nr. 04/2001. Fachbereich Informatik. Kaiserslautern (Universität Kaiserslautern) 2001.

[Met03] Metzger, Andreas: "Automatisierung der Entwicklung reaktiver Systeme durch Einsatz eines Produktmodells" Kolloquiumsvortrag. Institut für Informatik. Humboldt Universität zu Berlin. Januar, 2003. Berlin 2003.

[Met03a] Metzger, Andreas: "Requirements Engineering by Generator-Based Prototyping" Aus: Alt, Helena; Becker, Martin (Hrsg.): *Software Reuse: Requirements, Technologies and Applications. Proceedings*. International Colloquium of the SFB 501. Kaiserslautern. März, 2003. Kaiserslautern (Universität Kaiserslautern) 2003. S. 25–35.

[Met03b] Metzger, Andreas: "Konzeption und Analyse eines Softwarepraktikums im Grundstudium" Aus: Siedersleben, Johannes; Weber-Wulff, Deborah (Hrsg.): *Software Engineering im Unterricht der Hochschulen SEUH 2003*. Heidelberg (dPunkt Verlag) 2003. S. 41–48.

[Met04] Metzger, Andreas: "Feature Interactions in Embedded Control Systems" In: *Computer Networks*, 45. Jg. (2004), H. 5, S. 625–644.

[Met04a] Metzger, Andreas: "Effiziente Modellierung durch Automatisierung" Aus: Rumpe, Bernhard; Hesse, Wolfgang (Hrsg.): *Modellierung 2004*. Proceedings. Tagung Modellierung. Marburg, März, 2004. Lecture Notes in Informatics (LNI). P-45. Bonn (Köllen Druck+Verlag GmbH) 2004. S. 287–288.

[Met98] Metzger, Andreas: *Tool Supported Prototyping for the Specification and the Design of Building Automation Systems*. Diplomarbeit. Kaiserslautern (Universität Kaiserslautern) 1998.

[Met99] Metzger, Andreas: *An Interlink of Building Control System Prototypes and the Lighting Simulation Lumina*. Technischer Bericht. Fachbereich Informatik. Kaiserslautern (Universität Kaiserslautern) 1999.

[MeW03] Metzger, Andreas; Webel, Christian: "Feature Interaction Detection in Building Control Systems by Means of a Formal Product Model" Aus: Amyot, Daniel; Logrippo, Luigi (Hrsg.): *Feature Interactions in Telecommunications and Software Systems VII*. Proceedings Seventh International Workshop on Feature Interactions in Telecommunication and Software Systems, Ottawa, Kanada. Juni, 2003. Amsterdam (IOS Press) 2003. S. 105–121.

[MeZ03] Metzger, Andreas; Zimmermann, Gerhard (Hrsg.): *Modellierung Reaktiver Systeme: Ein Fallbeispiel*. SFB 501 Bericht 08/03. Sonderforschungsbereich 501. Technische Universität Kaiserslautern. Kaiserslautern 2003.

[Mil02] Milicev, Dragan: "Automatic Model Transformations Using Extended UML Object Diagrams in Modeling Environments" In: *IEEE Transactions on Software Engineering*, 28. Jg. (2002), H. 4, S. 413–431.

[MiM03] Miller, Joaquin; Mukerji, Jishnu: *Model Driven Architecture (MDA) Guide*. Version 1.0.1. Document Number: omg/2003-06-01. (Object Management Group) 2003.

[MMZ02] Mahdavi, Ardeshir; Metzger, Andreas; Zimmermann, Gerhard: "Towards a Virtual Laboratory for Building Performance and Control" Aus: Trappl, Robert (Hrsg.): *Cybernetics and Systems 2002*. Vol. 1. Proceedings 16th European Meeting on Cybernetics and Systems Research (EMCSR). Univeristät Wien. April, 2002. Wien 2002. S. 281–286.

[Mor99] Morand, Bernard: "Modeling: Is it Turning Informal into Formal?" Aus: Bezivin, Jean; Muller, Pierre-Alain (Hrsg.): *UML '98 - The Unified Modeling Language: Beyond the Notation*. Proceedings First International Workshop. Mulhouse, Frankreich. Juni, 1998. LNCS 1618. Heidelberg (Springer-Verlag) 1999. S. 37–48.

[MSS96] Middendorf, Stefan; Singer, Reiner; Strobel, Stefan: *JAVA Programmierhandbuch und Referenz*. Heidelberg (dpunkt Verlag) 1996.

[MSU02] Mellor, Stephen J.; Kendall, Scott; Uhl, Axel et al.: "Model-Driven Architecture" Aus: Bruel, Jean-Michel (Hrsg.): *Advances in Object Oriented Information Systems*. Proceedings OOIS 2002. Montpellier, Frankreich, September, 2002. LNCS 2426. Heidelberg (Springer-Verlag) 2002. S. 290–297.

[MTA99] Mellor, Stephen J.; Tockey, Steve; Arthaud, Rodolphe et al.: "An Action Language for UML: Proposal for a Precise Execution Semantics" Aus: Bezivin, Jean; Muller, Pierre-Alain (Hrsg.): *UML '98 - The Unified Modeling Language: Beyond the Notation*. Proceedings First International Workshop. Mulhouse, Frankreich. Juni, 1998. LNCS 1618. Heidelberg (Springer-Verlag) 1999. S. 307–318.

[MTB94] Morris, F.B.; Braun, J.E.; Treado, S.J.: "Experimental and Simulated Performance of Optimal Control of Building Thermal Storage" In: *ASHRAE Transactions*, 100. Jg. (1994), H. 1, S. 402–414.

[Nic98] Nicolau, Guillem B.: *Specification and Analysis of Weakly Hard Real-Time Systems*. Palma, Mallorca, Spanien 1998.

[NiJ99] Nissen, Hans; Jarke, Matthias: "Repository Support for Multi-Perspective Requirements Engineering" In: *Information Systems*, 24. Jg. (1999), H. 2, S. 131–158.

[OCL04] —: *Dresden OCL Toolkit*. Web-Site. 2004. `http://dresden-ocl.sourceforge.net/index.html`

[Ode98] Odell, James: *Advanced Object-Oriented Analysis & Design Using UML*. Cambridge, UK (Cambridge University Press) 1998.

[OdR99] Odell, James; Ramackers, Guus: "Martin/Odell Approach: A Formalization for OO" Aus: Zamir, Saba (Hrsg.): *Handbook of Object Technology*. Boca Raton, London, New York (CRC Press) 1999. S. 12-1–12-9.

[OFM94] Olsen, Anders; Faergemand, Ove; Moeller-Pedersen, Birger et al.: *Systems Engineering Using SDL-92*. Amsterdam, Niederlande (Elsevier Science B.V.) 1994.

[OMG01] —: *OMG Unified Modeling Language Specification*. Version 1.4. Document Number: formal/01-02-15. (Object Management Group) 2001.

[OMG02] —: *Common Object Request Broker Architecture: Core Specification*. Version 3.0. Document Number: formal/02-12-02. (Object Management Group) 2002.

[OMG02a]—: *Meta Object Facility (MOF)*. Version 1.4. Document Number: formal/02-04-03. (Object Management Group) 2002.

[OMG03] —: *OMG Unified Modeling Language Specification Version 1.5*. Document Number: formal/03-03-01. (Object Management Group) 2003.

[OMG03a]—: *XML Metadata Interchange (XMI) Specification*. Version 2.0. Document Number: formal/03-05-02. (Object Management Group) 2003.

[OMG04] —: *UML 2.0 OCL Specification*. Final Adopted Specification. Document Number: ptc/03-10-14. (Object Management Group) 2004.

[ONe98] O'Neil, Joe: *JavaBeans Programming from the Ground up*. Berkeley, Calif., USA (Osborne McGraw-Hill) 1998.

[PaQ95] Parr, Terence; Quong, Russell: "ANTLR: A Predicated-LL(k) Parser Generator" In: *Journal of Software Practice and Experience*, 25. Jg. (1995), H. 7, S. 789–810.

[PCW01] Parnas, David L.; Clements, Paul C.; Weiss, David M.: "The Modular Structure of Complex Systems" Aus: Hoffman, Daniel M.; Weiss, David M. (Hrsg.): *Software Fundamentals. Collected Papers by David L. Parnas*. Boston, Mass., USA (Addison-Wesley) 2001. S. 319–336.

[Pew04] Pews, Gerhard: "Software-Engineering im IT-Zeitalter" Vortrag sd&m. Industriekontaktmesse kontakt 2004. 25.05.2004. Universität Kaiserslautern. Kaiserslautern 2004.

[Pin02] Pinheiro, Francisco: "Requirements Honesty" Aus: Eberlein, Armin; Leite, Julio (Hrsg.): *International Workshop on Time-Constrained Requirements Engineering 2002* (TCRE'02). Proceedings. Essen, Deutschland, September 2002. 2002.

[PoB93] Pomberger, Gustav; Blaschek, Günther: *Grundlagen des Software Engineering. Prototyping und objektorientierte Software-Entwicklung*. München, Wien (Carl Hanser Verlag) 1993.

[Poe00] Poetzsch-Heffter, Arnd: *Konzepte objektorientierter Programmierung. Mit einer Einführung in Java*. Berlin, Heidelberg (Springer-Verlag) 2000.

[Poh96] Pohl, Klaus: *PRO-ART: Enabling Requirements Pre-Traceability*. Aachener Informatik-Berichte. 96-4. Aachen (RWTH Aachen, Fachgruppe Informatik) 1996.

[Por02] Porres, Ivan: *A Toolkit for Manipulating UML Models*. Technical Report No. 441. Turku, Finnland (Turku Center for Computer Science) 2002.

[PPS92] Pomberger, Gustav; Pree, Wolfgang; Stritzinger, Alois: "Methoden und Werkzeuge für das Prototyping und ihre Integration" In: *Informatik Forschung und Entwicklung*, 7. Jg. (1992), S. 49–61.

[Pri03] Prinz, Andreas: "SDL Time Extensions from a Semantic Point of View" Aus: Sherratt, Edel (Hrsg.): *Telecommunications and beyond: The Broader Applicability of SDL and MSC*. Proceedings 3rd International Workshop on SDL and MSC (SAM). Aberystwyth, Wales. Juni , 2002. LNCS 2599. Heidelberg (Springer-Verlag) 2003. S. 53–60.

[PSC01] Pulvermüller, Elke; Speck, Andreas; Coplien, James O. et al. (Hrsg.): *Feature Interaction in Composed Systems*. Proceedings. ECOOP 2001 Workshop #08. 15th European Conference on Object-Oriented Programming. Budapest, Ungarn. Juni, 2001. Universität Karlsruhe. Fakultät für Informatik. Interner Bericht Nr. 2001-14. Karlsruhe (Universität Karlsruhe) 2001.

[PSS01] Priolo, C.; Sciuto, S.; Sperduto, F.: *Efficient Design Incorporating Fundamentals Improvements for Control and Integrated Optimisation*. Final Report EDIFICIO Project. Conphoebus SpA. Contract No. JOE3 CT97 0069. 2001.

[QST99] Queins, Stefan; Schürmann, Bernd; Tetteroo, Torsten: *Bewertung des dynamischen Verhaltens von SDL-Modellen*. SFB 501. Bericht 9/99. Kaiserslautern (FB Informatik. Universität Kaiserslautern) 1999.

[QTB02] Queins, Stefan; Trapp, Mario; Brack, Torben et al.: *Spezifikation Floor 32*. Online Entwicklungsdokumente. Fachbereich Informatik. Kaiserslautern 2002. `http://wwwagz.informatik.uni-kl.de/d1-projects/ResearchProjects/Floor32/`

[Que02] Queins, Stefan: *PROBAnD – eine Requirements-Engineering-Methode zur systematischen, domänenspezifischen Entwicklung reaktiver Systeme*. Kaiserslautern (Universität Kaiserslautern) 2002. (= Schriftenreihe Informatik)

[QuM02] Queins, Stefan; Metzger, Andreas: "The PROBAnD Railway Crossing Specification" Beitrag zum SDL-2000 Design Contest, 3rd SAM (SDL And MSC) Workshop. Aberystwyth, Wales. Juni, 2002.
`http://sdl-forum.org/SAM_contest/`

[QuZ99] Queins, Stefan; Zimmermann, Gerhard: *A First Iteration of a Reuse-driven, Domain-specific System Requirements Analysis Process*. SFB 501. Bericht 13/99. Kaiserslautern (FB Informatik. Universität Kaiserslautern) 1999.

[RaB02] Rausch, Andreas; Broy, Manfred: "Evolutionary Development of Software Architectures" Aus: Ambra, R.; Beck, V.; Boiangiu, L. et al. (Hrsg.): *Technology for Evolutionary Software Development*. Proceedings NATO/RTO Information Systems Technology Panel (IST). Bonn. September, 2002 (NATO) 2002. S. 17-1–17-14.

[RAM92] Richardson, Debra J.; Leif Aha, Stephanie; O'Malley, T. Owen: "Specification-Based Test Oracles for Reactive Systems" Aus: — (Hrsg.): *ICSE 1992*. Proceedings 14th International Conference on Software Engineering. Melbourne, Australien. Mai, 1992. (ACM Press) 1992. S. 105–118.

[Reh04] Rehm, Dominik: *Automatisierte Durchführung von Experimenten auf der Basis von Softwaremodellen*. Diplomarbeit. Fachbereich Informatik. Technische Universität Kaiserslautern. Kaiserslautern 2004.

[Rei90] Reisig, Wolfgang: *Petrinetze: eine Einführung*. 2. überarbeitete und erweiterte Auflage. Berlin, Heidelberg (Springer-Verlag) 1990.

[Rem91] Rembold, Ulrich: *Einführung in die Informatik für Naturwissenschaftler und Ingenieure*. München, Wien (Carl Hanser Verlag) 1991.

[ReS97] Rekers, Jan; Schürr, Andy: "Defining and Parsing Visual Languages with Layered Graph Grammars" In: *Journal of Visual Languages and Computing*, 8. Jg. (1997), H. 1, S. 27–55.

[RGG03] Rößler, Frank; Geppert, Birgit; Gotzhein, Reinhard: "CoSDL - An Experimental Language for Collaboration Specification" Aus: Sherratt, Edel (Hrsg.): *Telecommunications and beyond: The Broader Applicability of SDL and MSC*. Proceedings 3rd International Workshop on SDL and MSC (SAM). Aberystwyth, Wales. Juni , 2002. LNCS 2599. Heidelberg (Springer-Verlag) 2003.

[Rie02] Riegel, Jan Peter: "Flexibler Entwurf von Gebäudesimulatoren" Aus: Tavangarian, G.; Grützner, R. (Hrsg.): *Simulationstechnik*. Proceedings ASIM 2002. Rostock. September, 2002. Erlangen (Gruner Druck GmbH) 2002. S. 576–578.

[Rie04] Riegel, Jan Peter: *Entwurf angepasster Gebäudesimulatoren*. Dissertation. Kaiserslautern (TU Kaiserslautern) 2004.

[Rif04] Rifkin, Jeremy: *Das Ende der Arbeit und ihre Zukunft. Neue Konzepte für das 21. Jahrhundert*. Frankfurt, New York (Campus Verlag) 2004.

[RJB99] Rumbaugh, James; Jacobson, Ivar; Booch, Grady: *The Unified Modeling Language Reference Manual*. Reading, Harlow, Menlo Park (Addison-Wesley) 1999.

[RoB87] Rombach, Dieter; Basili, Victor: "Quantitative Software-Qualitätssicherung. Eine Methode zur Definition und Nutzung geeigneter Maße" In: *Informatik Spektrum*, Jg. 1987, H. 10, S. 145–158.

[Rom03] Rombach, Dieter: "A Process Platform for Experience-Based Software Development" Aus: Alt, Helena; Becker, Martin (Hrsg.): *Software Reuse: Requirements, Technologies and Applications*. Proceedings. International Colloquium of the SFB 501. Kaiserslautern. März, 2003. Kaiserslautern (Universität Kaiserslautern) 2003. S. 47–57.

[RoV95] Rossi, Giancarlo; Visioli, Carlo: "Energy and Comfort in Office Buildings" Aus: — (Hrsg.): *Right Light Three*. Proceedings 3rd European Conference on Energy-Efficient Lighting. Newcastle, UK. Juni, 1995. (IAEEL) 1995. S. 11–18.

[RSN03] Roßbach, Peter; Stahl, Thomas; Neuhaus, Wolfgang: "Grundlegende Konzepte und Einordnung der Model Driven Architecture (MDA)" In: *Javamagazin*, Jg. 2003, H. 9, S. 1–4.

[Saa03] Saad, Alexandre: "Prototyping bei der BMW Car IT" In: *JavaSPEKTRUM*, Jg. 2003, H. 2, S. 49–53.

[SAL02] Sprinkle, Jonathan; Agrawal, Aditya; Levendovszky, Tihamer et al.: "Domain Model Evolution in Visual Languages Using Graph Transformations" Aus: Tolvanen, Juha-Pekka; Gray, Jeff; Rossi, Matti (Hrsg.): *OOPSLA 2002 2nd Workshop on Domain Specific Languages*. Proceedings. Seattle, Wash., USA. November, 2002. u. J.

[Sch88] Schürmann, Bernd: *SLANG (Simple Data Description Language). Sprachbeschreibung*. SFB 124. Bericht Nr. 22/88. Fachbereich Informatik. Universität Kaiserlautern. Kaiserslautern 1988.

[Sch99] Schnieder, Eckehard: *Methoden der Automatisierung. Beschreibungsmittel, Modellkonzepte und Werkzeuge für Automatisierungssysteme*. Braunschweig, Wiesbaden (Vieweg) 1999.

[Sch99a] Schütze, Martin: *Eine musterbasierte Methode zur domänenspezifischen Modellierung und Generierung von Softwarekomponenten*. Dissertation. Aachen (Shaker Verlag) 1999.

[Sei03] Seidewitz, Ed: "What Models Mean" In: *IEEE Software*, Jg. 2003, H. September/Oktober, S. 26–32.

[SeK03] Sendall, Shane; Kozaczynski, Wojtek: "Model Transformation: The Heart and Soul of Model-Driven Development" In: *IEEE Software*, September/Oktober. Jg. (2003), S. 42–45.

[Sel03] Selic, Bran: "The Pragmatics of Model-Driven Development" In: *IEEE Software*, September/Oktober. Jg. (2003), S. 19–25.

[SGJ02] Sunye, Gerson; Le Guennec, Alain; Jezequel, Jean-Marc: "Using UML Action Semantics for Model Execution and Transformation" In: *Information Systems*, 27. Jg. (2002), H. 6, S. 445–457.

[She01] Sheard, Tim: "Accomplishments and Research Challenges in Meta-programming" Aus: Taha, W. (Hrsg.): *Semantics, Applications, and Implementation of Program Generation*. Proceedings. Second International Workshop SAIG. Florenz, Italien. September, 2001. LNCS 2196. Heidelberg (Springer-Verlag) 2001. S. 2–44.

[She93] Shepperd, Martin: *Software Engineering Metrics Volume I. Measures and Validations*. London u.a. (McGraw-Hill Book Company) 1993.

[Sie04] Siedersleben, Johannes: *Moderne Softwarearchitektur. Umsichtig planen, robust bauen mit Quasar*. Heidelberg (dPunkt Verlag) 2004.

[SoC02] Sourrouille, Jean Louis; Caplat, Guy: "Checking UML Model Consistency" Aus: Kuzniarz, Ludwik; Reggio, Gianna; Sourrouille, Jean Louis et al. (Hrsg.): *UML 2002 Workshop on Consistency Problems in UML-based Software Development*. Proceedings. Dresden, Deutschland. September/Oktober, 2002. 2002.

[SPH01] Sunye, Gerson; Pennaneac'h, Francois; Ho, Wai-Ming: "Using UML Action Semantics for Executable Modeling and Beyond" Aus: Dittrich, Klaus R.;Geppert, Andreas; Norrie, Moira C. (Hrsg.): *Advanced Information Systems Engineering*. Proceedings. 13th International Conference. CAiSE 2001. Interlaken, Schweiz. Juni, 2001. LNCS 2068. Heidelberg (Springer-Verlag) 2001. S. 433–447.

[SPH02] Schätz, Bernhard; Pretschner, Alexander; Huber, Franz et al.: "Model-Based Development of Embedded Systems" Aus: Bruel, Jean-Michel (Hrsg.): *Advances in Object Oriented Information Systems*. Proceedings OOIS 2002. Montpellier, Frankreich, September, 2002. LNCS 2426. Heidelberg (Springer-Verlag) 2002. S. 298–311.

[SpH96] Sperschneider, Volker; Hammer, Barbara: *Theoretische Informatik: eine problemorientierte Einführung*. Berlin, Heidelberg (Springer-Verlag) 1996.

[SRP03] Streitferdt, Detlef; Riebisch, Matthias; Philippow, Ilka: "Details of Formalized Relations in Feature Models Using OCL" Aus: — (Hrsg.): *10th IEEE Symposium and Workshops on Engineering of Computer-Based Systems*. Proceedings. ECBS. Huntsville, Ala., USA. April, 2003 (IEEE Computer Society) 2003. S. 297–304.

[Ste04] Steimann, Friederich: "UML-A oder warum die Wissenschaft ihr eigene einheitliche Modellierungssprache haben sollte" Aus: Rumpe, Bernhard; Hesse, Wolfgang (Hrsg.): *Modellierung 2004*. Proceedings. Tagung Modellierung. Marburg, März, 2004. Lecture Notes in Informatics (LNI). P-45. Bonn (Köllen Druck+Verlag GmbH) 2004. S. 121–133.

[StK02] Steimann, Friedrich; Kühne, Thomas: "A Radical Reduction of UML's Core Semantics" Aus: Jezequel, Jean-Marc; Hußmann, Heinrich; Cook, Stephen (Hrsg.): *UML 2002 - The Unified Modeling Language: Model Engineering, Concepts and Tools*. Proceedings 5th International Conference. Dresden, Deutschland. September, 2002. LNCS 2460. Heidelberg (Springer-Verlag) 2002. S. 35–48.

[Stü91] von Stülpnagel, Alexander: "Repositories - Konzepte, Architekturen, Standards" In: *HMD Theorie und Praxis der Wirtschaftsinformatik*, 28. Jg. (1991), H. 161, S. 10–25.

[Tay99] Taylor, David A.: "The Keys to Object Technology" Aus: Zamir, Saba (Hrsg.): *Handbook of Object Technology*. Boca Raton, London, New York (CRC Press) 1999. S. 1-1–1-12.

[Tel03] —: *Abstract Data Type for Random Numbers*. Telelogic Tau SDL Suite 4.5 Online Documentation. 63 The ADT Library. 2003.

[Ter01] Terrasse, Marie-Noelle: "A Metamodeling Approach to Evolution" Aus: Balsters, Herman; De Brock, Bert; Conrad, Stefan (Hrsg.): *Database Schema Evolution and Meta-Modeling*. Proceedings 9th International Workshop on Foundations of Models and Languages for Data and Objects (FoMLaDO/DEMM 2000). Dagstuhl. September, 2000. Heidelberg (Springer-Verlag) 2001. S. 202–219.

[Tra93] Trauboth, Heinz: *Software-Qualitätssicherung. Konstruktive und analytische Maßnahmen*. München, Wien (Oldenbourg) 1993.

[Var04] Varro, Daniel: "Automated Formal Verification of Visual Modeling Languages by Model Checking" In: *Software and Systems Modeling*, Jg. 2004, H. 3, S. 85–113.

[VGP01] Varro, Daniel; Gyapay, Szilvia; Pataricza, Andras: "Automatic Transformation of UML Models for System Verification" Aus: Whittle, Jonathan (Hrsg.): *WTUML: Workshop on Transformations in UML*. Proceedings. ETAPS 2001 Satellite Event. Genova, Italien. April, 2001. 2001.

[VIS04] —: *VISEK. Virtuelles Software Engineering Kompetenzzentrum*. Web-Site. 2004/ `http://www.visek.de/`

[Völ04] Völter, Markus: "Modellgetriebene Softwareentwicklung" In: *OBJEKTspektrum*, Jg. 2004, H. 4, S. 14–20.

[Wal04] Walke, Christian: *Eine Fallstudie zur Modellierung eines Gebäudeautomationssystems*. Projektarbeit. Fachbereich Informatik. Technische Universität Kaiserslautern. Kaiserslautern 2004.

[WiD94] Wieringa, Roel; Dubois, Ericq: "Integrating Semi-formal and Formal Software Specification Techniques" In: *Information Systems*, 19. Jg. (1994), H. 4, S. 33–54.

[Wie00] Wiebicke, Ralf: *Utility Support for Checking OCL Business Rules in Java Programs*. Diplomarbeit. Lehrstuhl Softwaretechnologie. Institut für Software- und Multimediatechnik (SMT). Fakultät Informatik. Technische Universität Dresden. Dresden 2000.

[Wie98] Wieringa, Roel: "A Survey of Structured and Object-Oriented Software Specification Methods and Techniques" In: *ACM Computing Surveys*, 30. Jg. (1998), H. 4, S. 459–527.

[Wil98] Wiles, Anthony: The Tree and Tabular Combined Notation. A Tutorial. Online Dokument. (Telia Research) 1998.
`http://www.etsi.org/`

[WiM03] Wilson, Michael; Magill, Ewan: "An Environmental Model for Service Interaction In Home Networks" Aus: *Postgraduate Research Conference in Electronics, Photonics, Communications and Software (Prep) 2003.* Proceedings. Exeter, UK. April, 2003. 2003.

[Wir95] Wirth, Niklaus: "A Plea for Lean Software" In: *IEEE Computer*, 28. Jg. (1995), H. 2, S. 64–68.

[WiS00] Whittle, Jon; Schumann, Johann: "Generating Statechart Designs from Scenarios" Aus: — (Hrsg.): *22nd International Conference on Software Engineering.* Proceedings ICSE '02. Limerick, Ireland. Juni, 2000. (ACM Press) 2000. S. 314–323.

[Wit87] Witt, J.: "Zur Quantifizierung der Software-Qualität" In: *Informatik Spektrum*, Jg. 1987, H. 10, S. 121–122.

[WKC01] Wilkie, Ian; King, Adrian; Clarke, Mike et al.: *UML ASL Reference Guide. ASL Language Level 2.5.* Manual Revision C. Surrey, Großbritannien 2001.

[WOI04] Wedekind, H.; Ortner, E.; Inhetveen, R.: "Informatik als Grundbildung" In: *Informatik Spektrum*, 27. Jg. (2004), H. 2, S. 172–180.

[WUG03] Weis, Torben; Ulbrich, Andreas; Heihs, Kurt: "Model Metamorphosis" In: *IEEE Software*, September/Oktober. Jg. (2003), S. 46–51.

[Zim02] Zimmermann, Gerhard: "Efficient Creation of Building Performance Simulators Using Automatic Code Generation" In: *Energy and Buildings*, Jg. 2002, H. 34, S. 973–983.

[Zim03] Zimmermann, Gerhard: "Modeling the Building as a System" Aus: — (Hrsg.): *Building Simulation 2003: For Better Building Design.* Proceedings Eight International IBPSA Conference. Eindhoven, Niederlande. August, 2003. (IBPSA) 2003. S. 1483–1490.

[ZiM04] Zimmermann, Gerhard; Metzger, Andreas: "A Software Generation Process for User-centered Dynamic Building System Models" Aus: Dikbas, Attila; Scherer, Raimar (Hrsg.): *eWork and eBusiness in Architecture, Engineering and Construction.* Proceedings. 5th European Conference on Product and Process Modelling in the Building and Construction Industry. ECPPM 2004. Instanbul, Türkei. September, 2004 Leiden, London u.a. (A.A. Balkema Publishers) 2004. S. 195–202.

Sachverzeichnis

Symbole

A

F

G

H

I

J

K

L

M

Q

R

S

T

U

V

W

X

Z

Lebenslauf

Persönliche Daten:

Name:	Andreas Metzger
Geburtstag:	4. April 1973
Geburtsort:	Annweiler am Trifels
Eltern:	Brigitta Metzger, geb. Müsel, Großhandelskauffrau Michael Metzger, Maurermeister

Schulausbildung:

1979 – 1983	Grundschulen Rinnthal und Annweiler am Trifels
1983 – 1985	Realschule Annweiler am Trifels
1985 – 1992	Trifelsgymnasium Annweiler am Trifels Abschluss: Abitur (Note: sehr gut)

Wissenschaftlicher Werdegang:

Okt. 1992 – Nov. 1998	Studium der Informatik mit Nebenfach Wirtschaftswissenschaften an der Universität Kaiserslautern Abschluss: Dipl.-Inform. (Note: sehr gut)
Okt. 1995 – Nov. 1998	Wissenschaftliche Hilfskraft am Fachbereich Informatik der Universität Kaiserslautern
Dez. 1998 – Okt. 2004	Wissenschaftlicher Mitarbeiter in der Arbeitsgruppe „VLSI Design und Architektur“ (Prof. Dr. G. Zimmermann) und im Sonderforschungsbereich 501 „Entwicklung großer Systeme mit generischen Methoden“ am Fachbereich Informatik der Universität Kaiserslautern
März – Apr. 1999	6-wöchiger Forschungsaufenthalt am Department of Architecture, Carnegie Mellon University, Pittsburgh, USA